APPLYING TABU HEURISTIC TO WIND INFLUENCED,
IMUM RISK, AND MAXIMUM EXPECTED COVERAGE R

THESIS

Mark R. Sisson, Major, USAF

AFIT/GOR/ENS/97M

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GOR/ENS/97M-20

APPLYING TABU HEURISTIC TO WIND INFLUENCED, MINIMUM RISK, AND MAXIMUM EXPECTED COVERAGE ROUTES

THESIS

Mark R. Sisson, Major, USAF

AFIT/GOR/ENS/97M

AFIT/GOR/ENS/97M-20

THESIS APPROVAL

NAME: Mark R. Sisson, Mark, USAF **CLASS:** GOR-97M

THESIS TITLE: Applying TABU Heuristic to Wind Influenced, Minimum Risk, and Maximum Expected Coverage Routes

DEFENSE DATE: 27 February 1997

COMMITTEE:	NAME/TITLE/DEPARTMENT	SIGNATURE
Advisor	Glenn Bailey, Lieutenant Colonel, USAF Assistant Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology	[illegible]
Reader	Jack A. Jackson, Lieutenant Colonel, USAF Assistant Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology	[illegible]
Reader	William B. Carlton, Lieutenant Colonel, USA Adjunct Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology	[illegible]

APPLYING TABU HEURISTIC TO WIND INFLUENCED, MINIMUM RISK AND MAXIMUM EXPECTED COVERAGE ROUTES

THESIS

Presented to the Faculty of the Graduate School of Engineering
Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Operations Research

Mark R. Sisson, B. S., M. A.
Major, USAF

February 1997

Acknowledgments

I express my deepest gratitude to my family. Jeannie, Matthew, and Megan have contributed more to this effort than they will ever know. They persevered through the difficult times, always providing unfailing support. They endured long, lonely hours. I only hope I will be able repay their love.

I sincerely thank LtCol Glenn Bailey for his assistance during this research. He was always available for questions. He provided insight into problems that were at the edge of my ability. I am especially grateful for his support and encouragement.

I thank LtCol William B. Carlton whose C++ code and help made this project possible. Also, LtCol Jack A. Jackson made material contributions to this effort.

Mark R. Sisson

Table of Contents

List of Figures

Page

List of Tables

Page

Abstract

The purpose of this thesis is to provide Air Combat Command a method for determining the number of predator unmanned aerial vehicles (UAVs) required to cover a pre-selected target set given a risk assessment of the individual targets and a risk averse decision maker.

Extending previous research that employs reactive TABU search methods for deterministic vehicle routing problems, this thesis incorporates wind effects that can significantly alter the travel times for any given scenario. Additionally, it accounts for possible attrition by introducing minimum risk and expected number of targets covered to the objective function. The results of the TABU search and subsequent Monte-Carlo simulation gives the number of predator's required to cover a target set, identifies "robust" routes, and suggests routes that increase the expected number of targets covered while reducing losses.

I. Analysis Overview

I.1. Purpose and Outline of Theses

The purpose of this thesis is to provide Air Combat Command (ACC) a method for determining the number of predator unmanned aerial vehicles (UAVs) required to cover a pre-selected target set given a risk assessment of the individual targets and risk aversion of the decision maker. The proposed method incorporates winds, attrition, and assumes tasked coverage is provided for the target set. Tasked coverage is the time a UAV is directed to provide surveillance of a target. The proposed method also provides insight into minimum route and minimum risk route structures.

The capability and history of the predator UAV are given in Appendix 1. The notation is developed in section I.2. The background for this research is given in Section I.3 and Appendix 2. In Section I.4 the motivation for this work is outlined. Section I.5 explains the multiple homogenous vehicle traveling salesman problem time window constrained (mTSPTW) with the predator UAV incorporating winds effects model. Section I.6 describes the mTSPTW incorporating the decision makers' risk aversion. After developing the concept of maximum expected number of targets covered and minimum risk routes, attrition and coverage is evaluated in Section I.7. The model is validated in Appendix 3. Finally, limitations and suggested changes are discussed in Section I.8. Appendix 4 discusses code modifications to the TABU search.

I.2. Notation

The following notation will help in describing the methods used for solving wind influenced, minimum risk, and maximum coverage TABU heuristics.

R_r = total number of targets or nodes in a route r. (All routes together are a tour)

a_r = index first target in route r.

b_r = index last target in route r.

n_r = index of n^{th} target in route r such that $a_r \leq n_r \leq b_r$.

T = Number of routes.

r = route index $r=1...T$.

$Ps(i)$= Probability of survival at target or nodes i, $i= 1...N$.

$m_r = \prod_{i=a_r}^{b_r} Ps(i) \forall r \in 1...T$

$Pk(i)= 1- Ps(i)$

$MIN\ Ps = \underset{T}{Min}\{m_r\}$

$MAX\ Pk = 1\text{-}MIN\ Ps$

λ_{1a} *or* λ_{1b} = Penalty describing decision makers risk aversion for minimum risk routes.

λ_{2a} or λ_{2b} = Penalty describing decision maker's risk aversion for maximum expected targets covered routes.

$PEN_{1a} = -\lambda_{1a} * MIN\ Ps$

$PEN_{1b} = \lambda_{1b} * MAX\ Pk$

$PEN_{2a} = -\lambda_{2a} * A(N)$

$PEN_{2b} = \lambda_{2b} * (1/A(N))$

dz = travel time vector (proportional to distance).

dx = x component of vector dz.

dy = y component of vector dz.

o = acute angle between wind vector and dz

w = wind vector

q = modeled wind component, $0<=q<=1$

$A(T)$ = Average number targets covered.

A_r = Average number of targets covered for route r.

I.3. Background

The vehicle routing problem with maximum coverage is an extension of the classical general vehicle routing problem (GVRP). A general review from the literature for the general vehicle routing problem includes Eilon *et al.* (1971) and Bodin (1983). Within a traditional deterministic framework, the difficulty of solving these problems is well documented by Lenstra (1981), Savelsbergh (1992), Desrochers *et al.* (1992) and Dumas *et al.* (1995). This suggests the use of alternative heuristic methods that provide good - though not necessarily optimal - solutions in a reasonable period of time (Glover 1989, 1990a, 1990b). This difficulty is further compounded by introducing expected coverage due to the presence of probabilistic UAV loss rates at one or more targets. Therefore, we selected the TABU search heuristic as a solution method robust enough to allow for additional objective function components (Laporte 1992) (see Appendix 2).

The TABU search (Glover 1989, 1990a, 1990b) attempts to avoid becoming stuck in local optima by exploiting memory and data structures to prevent returning to previously examined solutions. Reactive TABU search incorporates the basic parameters and memory structures of the basic TABU search. In addition, the heuristic automatically adjusts search parameters based on the quality of the search. The reactive TABU search developed by Carlton (1995) is the "engine" used in this thesis, with modifications to incorporate winds and risk aversion for predator unmanned aerial vehicles (UAVs).

Figure 1 summarizes our overall approach with an emphasis on three areas. First, two inputs distinguish this from previous TABU/VRP algorithms -- winds and probability of survival. Second, the two significant outputs are the maximum coverage and worst case route structure. Finally, a Monte-Carlo simulation evaluates the proposed route structures to more accurately characterize the distribution of vehicle losses and provided surveillance time of targets.

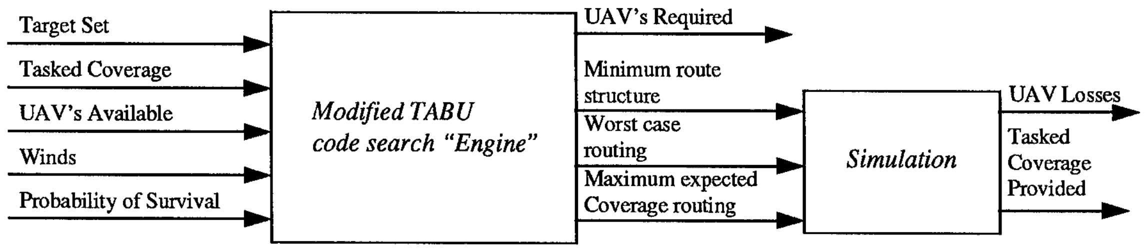


Figure 1. Analysis Overview

I.4. Motivation

The primary motivation of this thesis is to give decision makers and operators a tool that provides timely insight and high quality analysis of possible predator UAV operations without having to solve large, intractable integer programming problems. ACC wants to know how many predator UAV to allocate to a target set; this is equivalent to defining a set of routes since one UAV is assigned to each route. The TABU search was selected to provide these routes; however, a simple minimum distance solver does not account for the slow speed and vulnerability of the predator. Therefore, these variables were included into the TABU heuristic. This provides answers at the tactical level to the following questions.

1. How many predators should we send out today?

2. What are some suggested routes?

3. What routes are insensitive to wind changes?

Furthermore a Monte-Carlo post-optimization simulation answers the following:

1. What are the overall expected losses?

2. How much of the tasked coverage is expected to be accomplished?

In summary, this analysis provides a mechanism to the decision maker to operationally model attrition and effect of winds.

I.5. *Multiple homogenous vehicle Traveling Salesman Problem time window constrained including winds (mTSPTWIW)*

Incorporating winds into the model requires vector addition of wind vector to the flight time vector. (Figure 2)

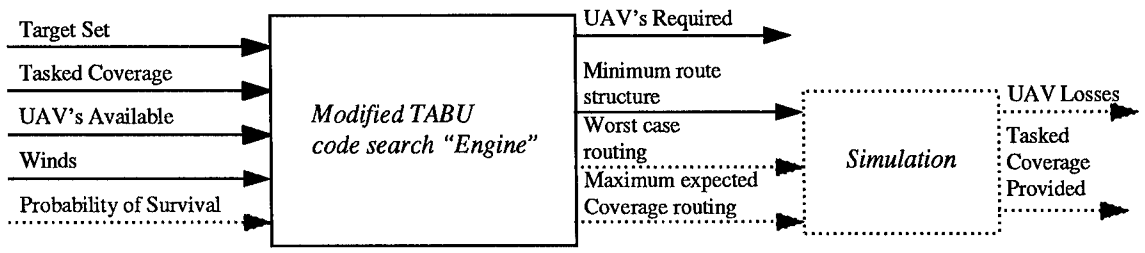


Figure 2. Winds in TABU heuristic

The model assumes winds are proportional to flight time. The longer the UAV is aloft the greater the resulting wind vector. These wind vectors modify flight time requirement to targets, and may be asymmetric. This is what we would expect since tailwind components would shorten and headwind components would lengthen a flight. Figure 3 graphically shows the use of vectors to model winds.

Modeling winds requires that we first compute the length of time *dz* for the no wind vector by $dz=(dz^2+dy^2)^{.5}$. Then, the appropriate wind vector to be modeled is selected as proportional to *dz'*- or *dz'*+. Finally, the law of cosines computes vectors *dz'*- and *dz'*+. For example letting $w=.2*dz-$ and angle $o = atan(dy/dx)$ then $dz'-=(dz^2+w^2-2*dz*w*cos(o))^{.5}$ (see Figure 3). The appropriate *dz'*- and *dz'*+ vectors are then incorporated into the heuristic.

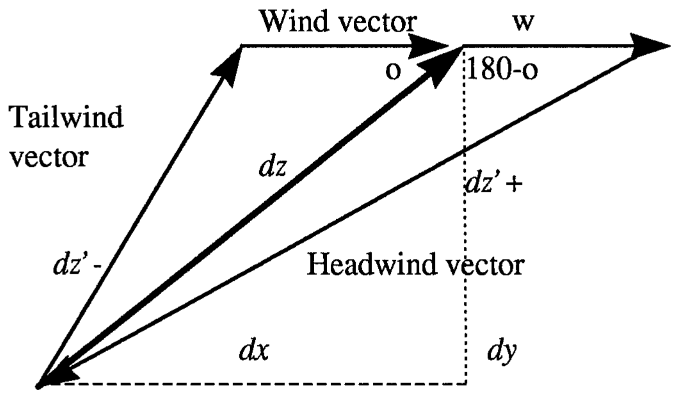


Figure 3. Wind influenced vectors

A data set QARI, shown in Table 1, was selected for modeling. The predator UAVs are time constrained in that they must return within 24 hours, and generally are tasked to provide 6 hours coverage per target. Additionally, there are no on-station time windows. Seven predators are made available each using an endurance speed of 70 nm/hr.

Table 1. QARI data set

	X Coordinate (min)	*Y Coordinate (min)*	*Early time window (min)*	*Late time window (min)*	*Surveillance time (min)*	*Probability of survival*
Depot	199.714	31.714	0	1440	0	1.0
1	24.857	192.857	0	1440	360	0.9
2	68.571	224.571	0	1440	360	0.9
3	128.571	175.714	0	1440	360	0.9
4	134.571	150.857	0	1440	360	0.9
5	183.428	124.287	0	1440	360	0.9
6	207.428	97.714	0	1440	360	0.9
7	247.714	133.714	0	1440	360	0.8
8	247.714	84.857	0	1440	360	0.9
9	271.714	130.285	0	1440	360	0.9
10	289.714	114.000	0	1440	360	0.9
11	266.571	160.286	0	1440	360	0.9
12	262.285	145.714	0	1440	360	0.8
13	258.000	103.714	0	1440	360	0.9
14	305.142	126.000	0	1440	360	0.9
15	311.142	102.000	0	1440	360	0.9
16	294.000	69.428	0	1440	360	0.9
17	311.142	71.142	0	1440	360	0.9
18	311.143	62.571	0	1440	360	0.9

Figure 4 shows the no wind risk indifferent route structure where seven UAVs are used. This solution of QARI is feasible since all tasked coverage is met. The total travel time for all seven predator UAV's is 8543.4.

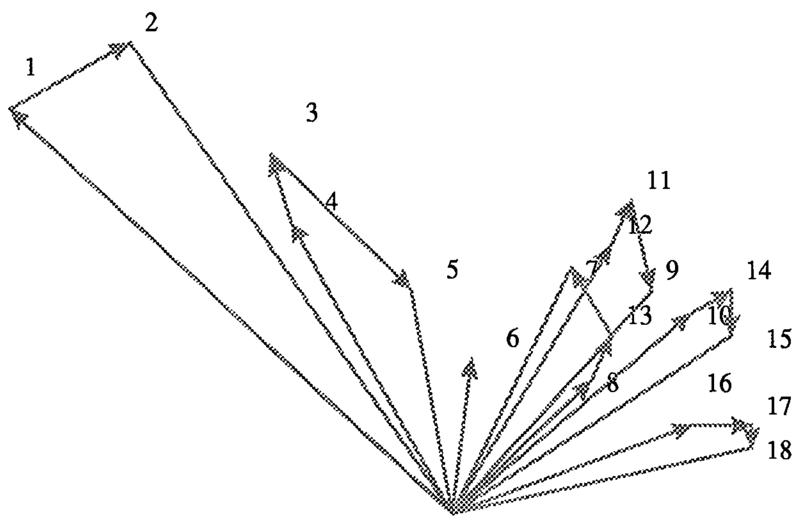


Figure 4. No wind QARI solution

The solution in Figure 4 is also the route structure for QARI with winds 14 nm/hr from the south. However, flight time increases by 132 minutes to 8675.8. Incorporating a 14 nm/hr westerly wind does change the route structure from Figure 4 to Figure 5. The solution is feasible and all tasked coverage is accomplished, with a total travel time of 8632.3 (an increase over the no-wind scenario by 88.9 minutes). Note that the routes in bold are routes different from the routing without winds.

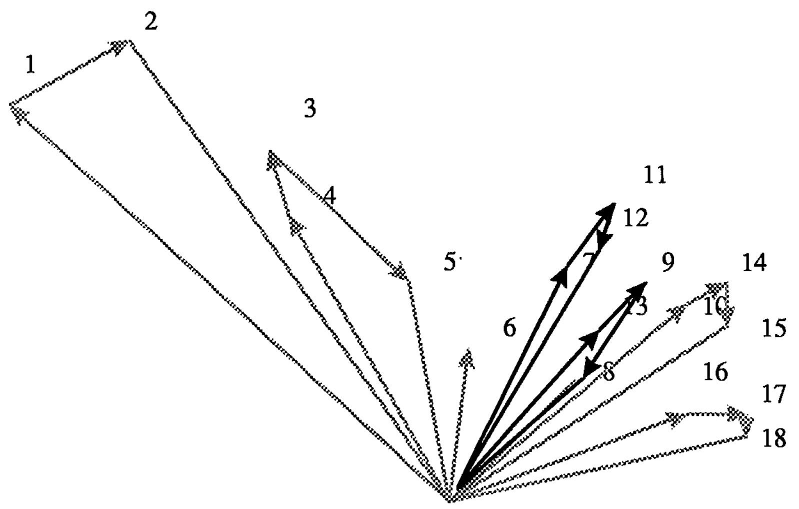


Figure 5.QARI 14 NM/HR WESTERLY WINDS

In conclusion, seven predators are required for a feasible solution to the QARI target set. The computed required predators are the minimum number of predators that produce a feasible solution.

I.6. mTSPTW incorporating Risk Aversion

The slow speed, lack of radar warning receivers, and the fact that intelligence is usually located with air defense sites, make the predator especially vulnerable. These factors are inherent limitations of operations and predator construction. To provide the decision maker more options, we introduce four separate penalty functions for incorporating UAV loss rates in the objective function – two for worst case and two for expected number of targets covered. This provides the decision maker with a choice of different routes. The decision maker can then choose which tour best accomplishes his or her goals. (Figure 6)

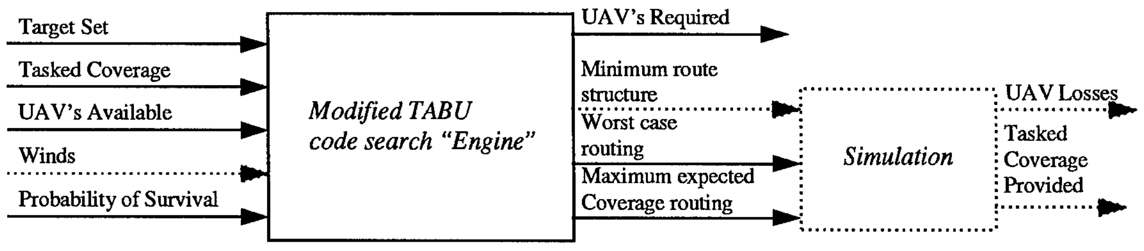


Figure 6. Risk into the TABU heuristic

I.6.1 Worst-Case Routing

In the worst case routing, first the lowest probability of survival or expected coverage for a target set in all routes *r* is computed. For any route *r* the probability of surviving the route is

$$\prod_{i=a_r}^{b_r} Ps(i) = m_r \, \forall r \in 1 \ldots T$$

The lowest probability of survival from among all routes is then

$$\underset{T}{Min}\{m_r\} = MIN\ Ps$$

Thus, the highest probability of kill from among all routes is

$$1 - MIN\ Ps = MAX\ Pk$$

Then, penalties are computed in the TABU objective function are $PEN_{1a} = -\lambda_{1a} * MIN\ Ps$ or $PEN_{1b} = \lambda_{1b} * MAX\ Pk$, where λ_{1a} *or* λ_{1b} is chosen to reflect the decision makers risk aversion. Empirical observations suggest that PEN_{1a} does a good job for problems less than 20 targets, which may be due to the intensification properties of the heuristic. Conversely, PEN_{1b} appears to work better on large problems. Thus, the algorithm perturbs the routes by swapping targets around, searching for a tour with an overall increase in *Ps(i)*.

Applying this procedure to the target set QARI gives the route structure in Figure 7. This example is based on nodes 7 and 11 having a probability of survival of .8, while all others have a probability of survival of .9. Figure 7 shows how risk aversion changes the route structure from Figure 4. The high threat target sets are assigned to routes with two targets as opposed to three (in bold). Intuitively this makes sense in that such re-routing reduces the chances of UAVs being lost on these routes.

Table 2. QARI risk aversion

QARI *data set*	$\lambda 1a$	*Number of UAV's*	*Worst Ps(i) in tour*	*Travel Time*
No Risk Aversion	0	7	.648	8543.4
Risk Aversion	2500	7	.72	8743.7

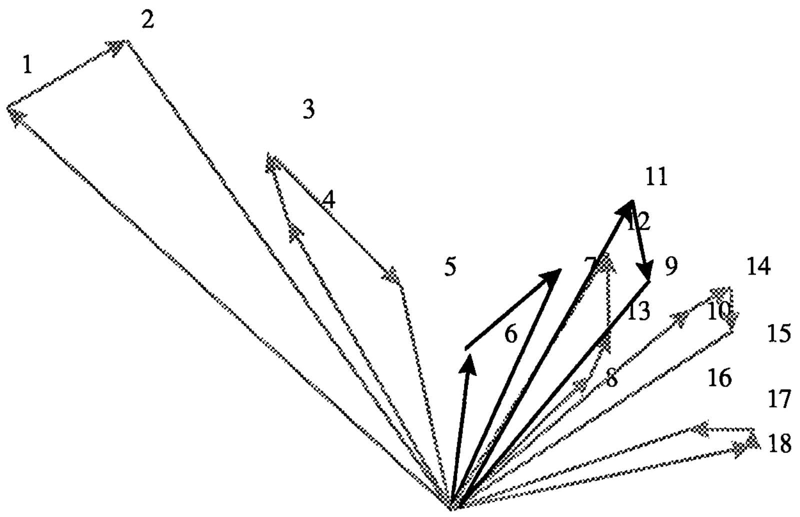


Figure 7. QARI risk aversion

1.6.2 Maximum Number Expected Target Coverage Routing

This algorithm calculates a penalty function based on the expected number of targets covered. Since expected coverage of any single target equals its probability of survival, then for target n_r in route r such that $a_r \leq n_r \leq b_r$

$$\prod_{i=a_r}^{n_r} Ps(i)$$

computes the expected number of times target n_r will be covered. (This is because the random variable is one for each target.) For instance, assume the predator travels from target 1 to 2 to 3; and, Ps(1)=.9, Ps(2)=.8, and Ps(3)=.7. Assuming independence, target 1 is covered 90%; target 2 is covered .9*.8=.72, or 72% of the time; and, target 3 is .9*.8*.7=.504, or 50.4% of time. The expected coverage A_r for route r is then computed by

$$\sum_{n_r=a_r}^{b_r} \prod_{i=a_r}^{n_r} Ps(i) = A_r$$

Continuing with the previous example, if the routing through targets 1-3 constitutes the entire tour, then A(N) = .9+.72+.504 = 2.12. For multiple routes in a tour, the expected coverage for the tour is computed by

$$\sum_{r=1}^{T} A_r = A(T).$$

Then $PEN_{2a} = -\lambda_{2a}*A(T)$ or $PEN_{2b} = \lambda_{2b}*(1/A(T))$ is used as a penalty in the TABU heuristic objective function. Again, empirical observations suggest that PEN_{2a} appears to work better for smaller problems while PEN_{2b} works better on larger problems.

There are several tendencies we would expect the TABU search to possess given the introduction PEN_{2a} *or* PEN_{2b} and disregarding time window constraints. First, as we increase the risk penalty, we expect the TABU heuristic to “re-sequence” existing targets on routes from greatest to lowest probability of survival. Second, when the penalty gets sufficiently high alternative routes appear. Finally, in the extreme a single UAV should be assigned to each target.

Changing the probability of survival for targets in the QARI data set to one that more accurately reflects a realistic scenario gives the data listed in Table 3. When the TABU heuristic is run with a penalty of $\lambda_{2a} = 2500$, the *A(T)* of the route jumps from 15.74908 to 16.12718. In this case the expected coverage is increased by reversing the routes, and with no increase in distance traveled.

Table 3. QARI Ps(i)

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
.99	.99	.95	.92	.80	.99	.80	.95	.99	.99	.99	.80	.99	.80	.70	.99	.99	.99

Applying a penalty of $\lambda_{2a} = 5000$ changes the route structure and increase distance flown (Figure 8). The expected nodes covered or *A(N)* is 16.07263.

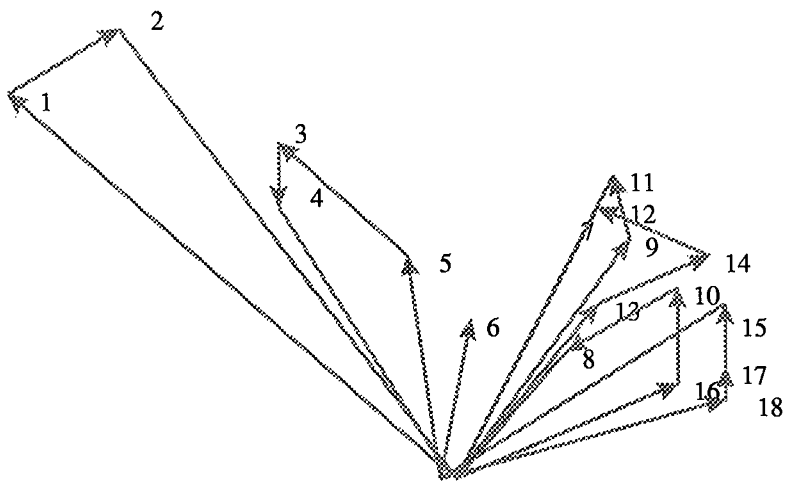


Figure 8. QARI Maximum number of Expected target coverage

I.7. Evaluation

A data set called NARI was developed (Table 3) to see how the heuristic handled more complex problems. The data set for NARI is shown in Table 4. Once again, the predator UAVs are time constrained in that they must return within 24 hours, and provide 6 hours coverage per target. In this model there are no on-station time windows. Predator endurance speed of 70 nm/hr is assumed.

Table 4. NARI data set

	X Coordinate	*Y Coordinate*	*Early time window*	*Late time window*	*Surveillance time*	*Probability of survival*
Depot	100.286	64.286	0	1440	0	1.0
1	7.714	381.429	0	1440	360	0.9
2	55.714	360.000	0	1440	360	0.8
3	81.429	351.429	0	1440	360	0.9
4	58.286	342.857	0	1440	360	0.6
5	65.143	325.714	0	1440	360	0.9
6	34.286	327.429	0	1440	360	0.8
7	70.286	296.571	0	1440	360	0.9
8	27.429	291.429	0	1440	360	0.8
9	93.429	297.429	0	1440	360	0.9
10	48.000	280.286	0	1440	360	0.8
11	76.286	269.143	0	1440	360	0.9
12	120.000	274.286	0	1440	360	0.8
13	160.286	291.429	0	1440	360	0.9
14	100.286	251.143	0	1440	360	0.8
15	114.000	216.000	0	1440	360	0.9
16	205.714	234.000	0	1440	360	0.8
17	104.571	219.429	0	1440	360	0.9
18	144.000	220.286	0	1440	360	0.8
19	126.857	203.143	0	1440	360	0.9
20	231.429	217.714	0	1440	360	0.8
21	292.286	191.143	0	1440	360	0.9
22	181.714	145.714	0	1440	360	0.8
23	200.571	140.571	0	1440	360	0.9
24	291.429	137.143	0	1440	360	0.6
25	214.286	121.714	0	1440	360	0.9
26	248.571	92.571	0	1440	360	0.6
27	274.286	82.286	0	1440	360	0.9
28	291.429	78.857	0	1440	360	0.8
29	332.571	82.286	0	1440	360	0.9
30	349.714	80.571	0	1440	360	0.8
31	377.143	84.000	0	1440	360	0.9
32	375.429	99.429	0	1440	360	0.8
33	385.714	111.429	0	1440	360	0.9
34	402.857	115.714	0	1440	360	0.8
35	404.571	106.286	0	1440	360	0.9
36	396.000	94.286	0	1440	360	0.8
37	432.000	92.571	0	1440	360	0.9
38	437.143	70.286	0	1440	360	0.8
39	447.429	43.714	0	1440	360	0.9
40	472.286	33.429	0	1440	360	0.8

Before addressing the NARI scenario, the number of iterations to adequately search the solution space was explored. For the NARI data set a graph comparing solution results to iterations is given in Figure 9. Given the diminishing returns of solution improvement for an additional number of runs, two thousand iterations was determined to be adequate for the minimum risk and maximum coverage models.

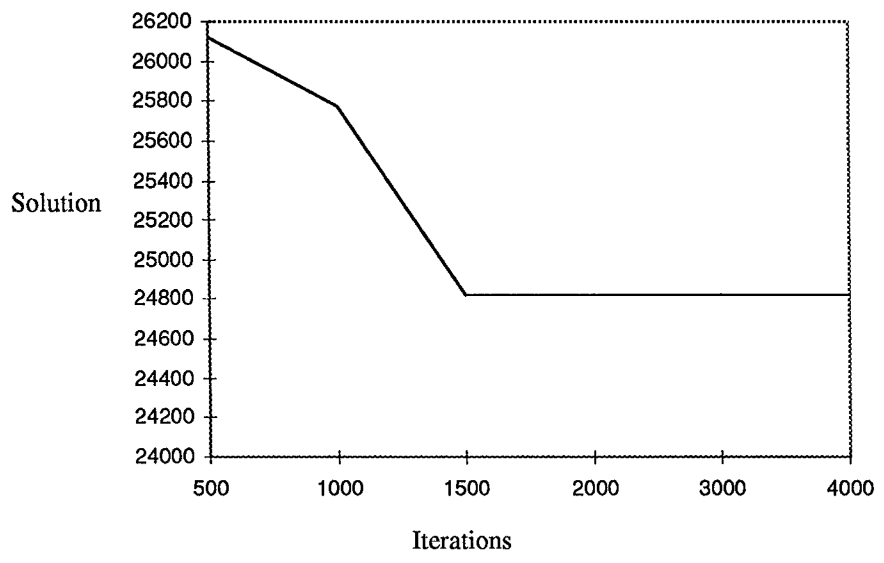


Figure 9. Solution versus iterations

The no wind solution with 20 allowed UAV's has a travel time of 24,745.4, and is shown in Figure 10. The solution for 14 nm/hr westerly wind is 25,799.5 and requires 20 predators, while the solution for 14 nm/hr southerly wind is 26,518.3 requiring 22 predators. The required predators for all scenarios are the minimum necessary to produce a feasible solution.

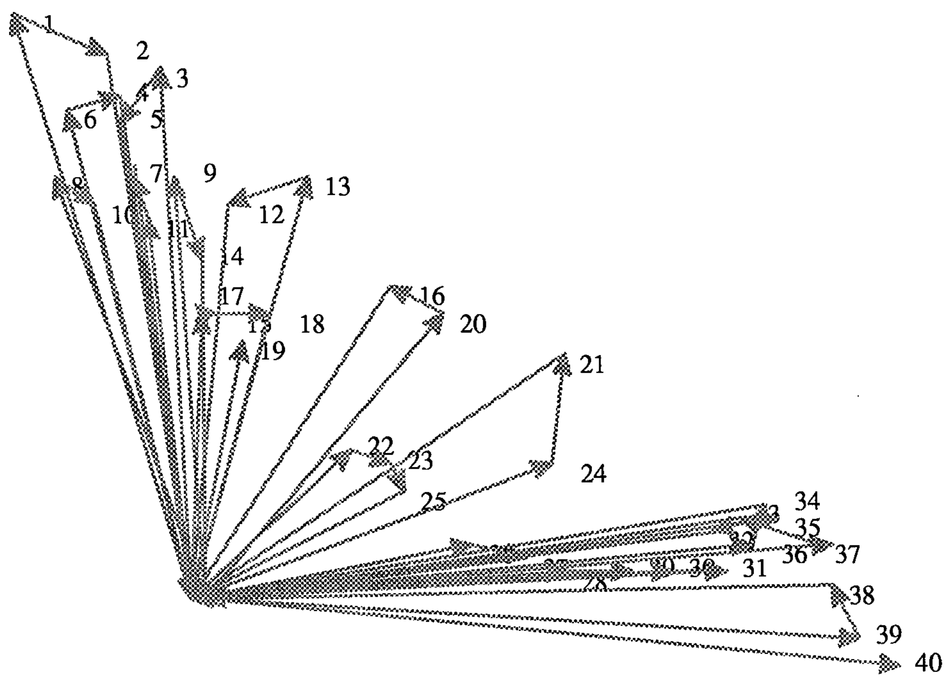


Figure 10. NARI no wind solution

The proposed solution for minimum risk coverage and maximum expected number of targets is shown for NARI in Figure 11 and 12. The penalties λ_{1b} and λ_{2b} are set to 100,000 each for these solutions, thus being large enough to override minimum route structure.

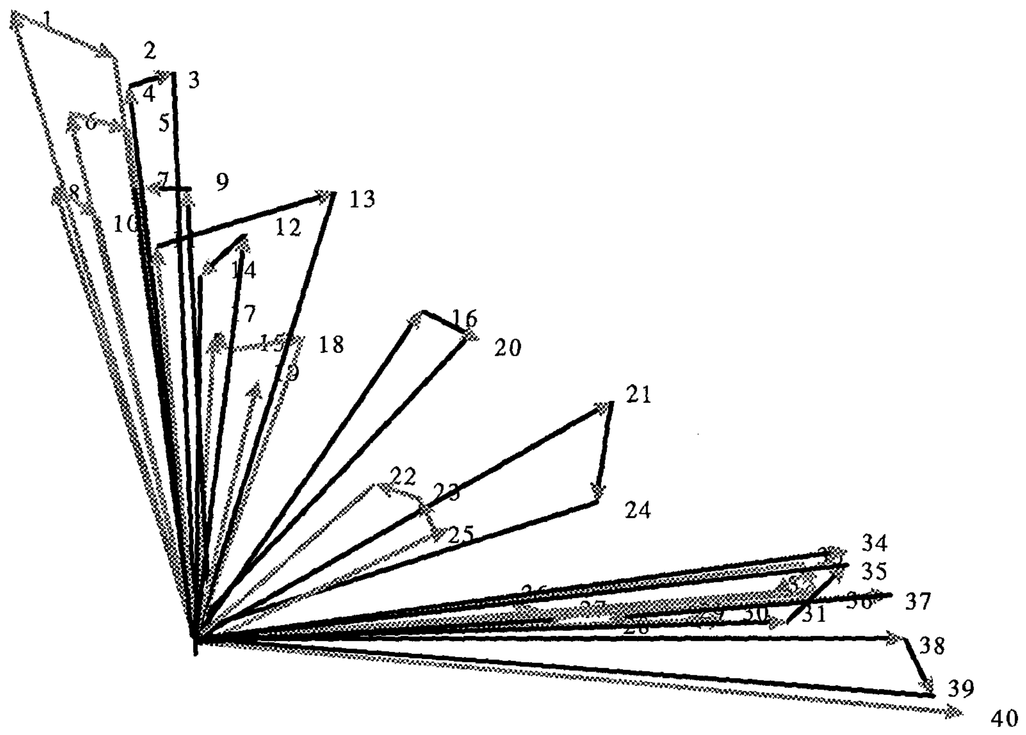


Figure 11. NARI **Minimum risk coverage**

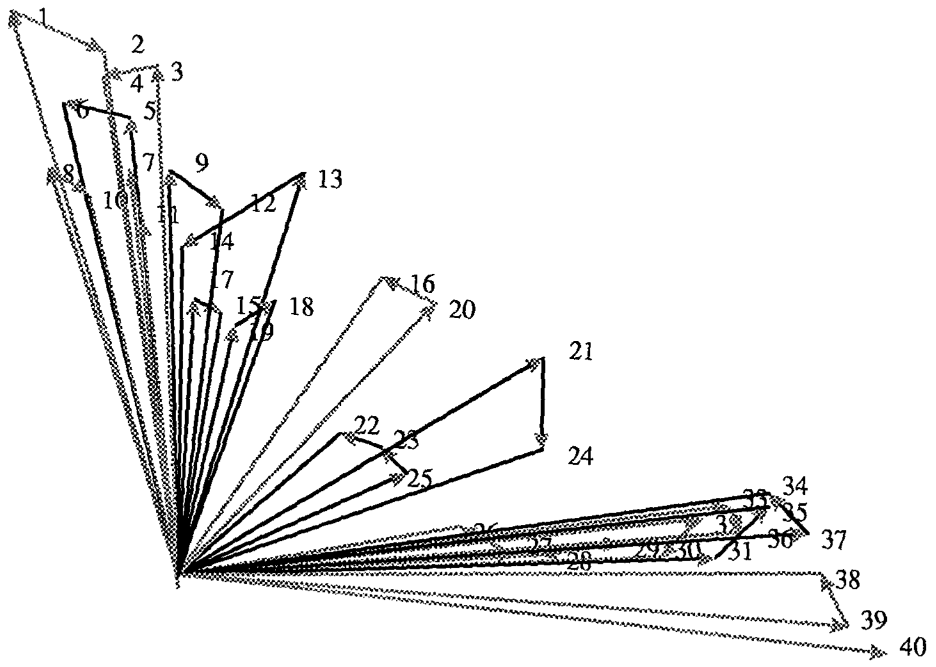


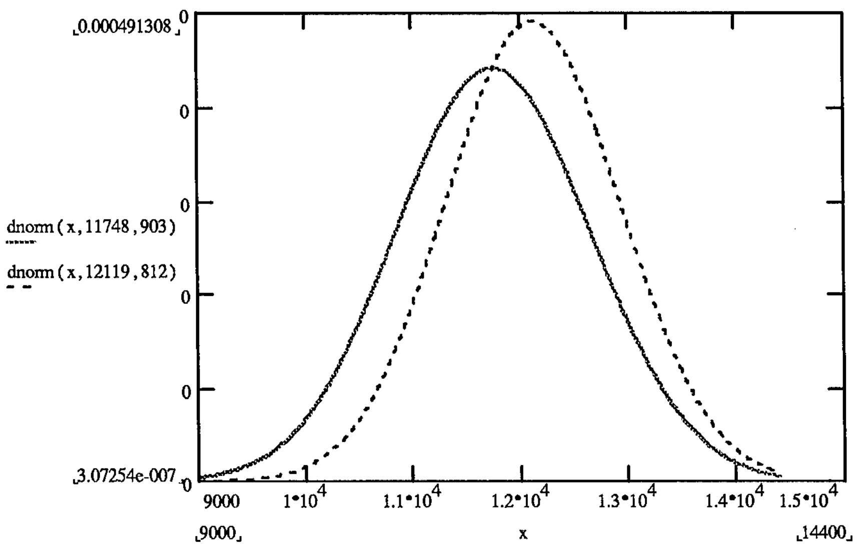
Figure 12. NARI **Maximum expected targets covered routes**

One way to calculate expected coverage is to simply take *A(T)* and multiply it by time of coverage. Using this technique, for the NARI problem the provided coverage is 30.446*360=10,960.56 minutes. Note that this is different than what is shown in Table 5 because this technique does not account for any partial coverage provided by a predator before it is shot down. For instance, if a predator is downed two hours into providing datalink coverage over a target, this time is not captured. To account for this, and to model predator losses, a Monte-Carlo spreadsheet simulation was developed. Table 5 compares the simulation results of the three models based on 500 samples.

Table 5. NARI Comparisons

Objective	λ_{2b}	*Number of UAV's*	*Expected Tasked Coverage*	*Standard Deviation of Tasked Coverage*	*Expected Predator Losses*	*Standard Deviation of Losses*	*Flight Time*
Minimum Travel time	0	20	11,748.27	903.26	3.228	1.5194	24,745.40
Minimum Risk Route	100000	20	11,819.49	895.8	2.998	1.5371	24,838.30
Maximum targets covered	100000	20	12,119.53	812.56	2.214	1.3431	24,758.30

The "maximum expected targets covered" model produces the highest expected tasked coverage with lowest expected losses. Note that the increased flight time of 12.9 minutes of the "maximum targets covered" criteria produces an increased expected tasked coverage of 371.26 over the minimum travel time. Also, the variance of the empirical distribution for "maximum targets covered" decreases from "minimum travel" time. The empirical distribution for tasked coverage and vehicle attrition is graphically depicted in Figures 13 and 14.



——————— Minimum Distance Route

- - - - - - - - Maximum Expected Coverage Route

Figure 13. Empirical distribution of tasked coverage

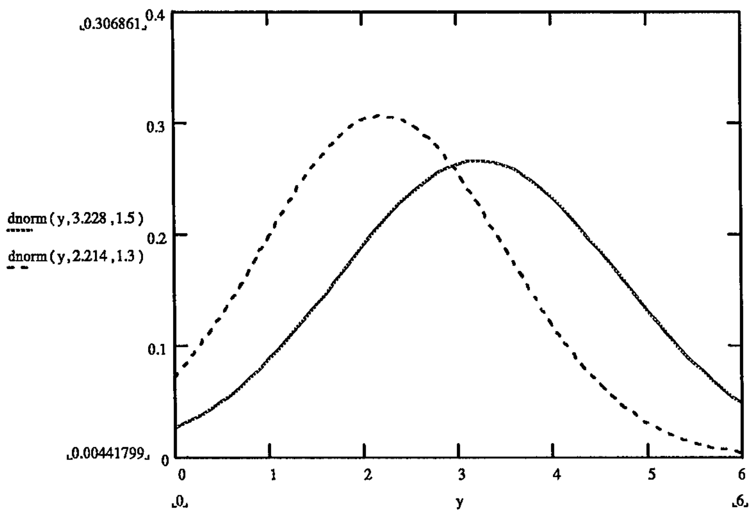


Figure 14. Empirical distribution of UAV losses

I.8. Areas of further research and study

The algorithm, as it stands, only takes a snapshot of one day predator UAV operations. However, a more realistic scenario involves a dynamic 30 day operation. In such an environment the TABU heuristic could be run for every day; however, some vehicles land after 9 hours and others fly up to 24 hours. It would be difficult to perform maintenance if a vehicle has several concurrent long endurance flights. A packing heuristic allows for maintenance and efficient multiple scheduling of the predator UAVs Taillard *et al.* (1996). Another, less elegant, way to handle multiple sorties is to "trick" the algorithm, by relaxing time window constraints for the depot; introducing maintenance nodes located at the same coordinates as the depot; introducing time constraints that force the vehicles to return to "maintenance" nodes; and, introducing a service time penalty to preclude one vehicle servicing all "maintenance" nodes. Still another interesting question is what combination of vehicles provide for maximum coverage and minimum loss within available resources? Finally, what properties of the problem structure and λ result in different route structures.

In summary, the TABU search produces near optimal results in a fraction of the time required by integer programming. This allows for quick analysis of minimum route structures and insight into resources required to cover those route structures. Also, the methodology provided gives a decision maker a set of alternate routes. From these routes a decision maker can easily choose a plan that conforms to his or her risk aversion within resources available.

II. Appendix 1

II.1. General Issues

Unmanned Aerial Vehicles (UAV) are not new. In fact, the United States has employed UAVs for reconnaissance purposes since the Korean War. Later, during the Southeast Asia conflict DC-130's launched "special purpose aircraft".

The Persian Gulf war, and advances in technology, have caused the services and Joint Staff to once again consider UAVs a viable weapon system. A 1994 study by Defense Agency Reconnaissance Office (DARO) notes that the United States deployed more than 85% of its available assets during the Gulf war, and concludes that "It is obvious that future requirements will exceed our current capability to collect, process and exploit information (Hewish 1995)". In response, DARO launched the Advanced Concept Technology Demonstration (ACTD) Program. The initial ACTD program started with the predator Tier II Medium Altitude Endurance (MAE) Vehicle. The 30 Month ACTD involves 10 general Atomic Air Vehicles, sensor payloads, and associated ground based equipment.

Another lesson learned during the Persian Gulf war is that the Air Force needs a diverse family of Unmanned Aerial Vehicles, and not one all-purpose model. Smaller, target spotting, tactical UAVs would be easier to operate near the front lines under the control of Corps or Division Commanders. Larger, longer endurance unmanned vehicles like the predator could take off far from the battlefield yet patrol large areas for the Joint

Task Force or Theater Chiefs. Therefore, UAVs are grouped into four operational categories: maneuver range, tactical range, medium range, and endurance (Pearson 1995).

Along with the Persian Gulf war, the advances in technology has driven the acquisition of UAVs. UAVs are being developed rapidly using available technology. The international market for UAVs is one of the rapidly expanding aerospace sectors. One of the key technological advances is the science of robotics. UAVs are much more autonomous than the autopilot equipped RPV's of the past. UAVs can now be programmed to fly to wait points, change altitude, and continue to their next target on their own. Quoting Air Force Chief of Staff Gen. Fogleman "UAVs hold great promise to perform many theater reconnaissance operations—from surveillance to targeting and bomb damage assessment (SAF 1996)".

The Department of Defense intends to spend $200M per year for the foreseeable future on UAV research and development (Grier 1996). The capabilities of UAV's merge with the Pentagon's emerging "information dominance". These unmanned vehicles can be lost with minimum political and military impact, as reflected in their relatively low per unit cost. In the United States, where public opinion wants military operations to be carried out with the least amount of casualties, UAVs offer this distinct advantage. Other advantages include the following potential applications (Pearson 1995): near real time targeting and precision strike support; near real time combat assessment; electronic order of battle gathering (EOB); bomb damage assessment (BDA); intelligence preparation of the battlefield; special operations; blockade and quarantine enforcement; sensitive reconnaissance operations (SRO); humanitarian aid; United Nation treaty monitoring; counterdrugs; single integrated operations plan (SIOP); and, communications relay.

Air Combat Command is considering a force of up to 90 UAVs of various types. Supporting this need, the predator contract was awarded in January 1994, and the first aircraft was flying in July 1994 (Grier 1996). In July 1995, predators deployed to Europe to support contingency operations in Bosnia. The deployment demonstrated over the horizon (OTH) control of UAVs under combat conditions. For example, predators watched suspected mass grave sites and documented any attempt to tamper with the bodies.

II.2. Predator UAV

The predator is a derivative of the successful Gnat 750. The Gnat was designed to provide long range, long dwell, near-real-time imagery intelligence (IMINT). It will be the operational workhorse for endurance vehicles until at least 1998. The UAV system includes air vehicle (predator), ground control station (GCS), sensor payloads, data links, ground support equipment and trained personnel. The GCS is a 30x8x8 ft triple axle trailer, which includes an uninteruptable power supply (UPS) and environmental control station (ECS). Stations in the GCS include mission planning, data exploitation station, air vehicle operator station, and payload operator station.

A fully deployable MAE UAV system consists of four predators, a ground control station, support equipment, a Trojan spirit II van, and 68 personnel. For deployment this configuration will require four C-130's or two C-141's. This configuration of equipment and personnel supports one continuos orbit for the predator. The Tier II MAE will follow the main operating base (MOB), forward operating location (FOL) concept. This means the tasked assets will be sent to the area they are needed and return after their commitment is complete.

Table 6. Predator Statistics

Maximum altitude	25,000 ft
Maximum endurance	40+ hours
True Air Speed	60-129 knots
Cruise Speed	70 knots
Radius	500 Nm
Sensors	SAR, EO, IR
Thrust	85 Hp
Length	26.7 ft
Width	3.7 ft
Navigation System	GPS, INS
Survivability Measures	None
Payload	450 lbs

The predator air vehicle is intended to be operated from a prepared strip (including grass and dirt runways). The predator is the UAV of choice when operational plans require watching a small area for days or weeks. The reason UAVs are the top choice is that overheads have windows of surveillance, while manned vehicles have endurance and cost constraints. Predators also have the ability to look into valleys and behind hills, which high altitude endurance (HAE) UAV's do not.

Sensors for the predator include electro-optical (EO)/(IR), which are two daylight video cameras and an IR camera. The EO/IR sensors provide enhanced resolution at lower altitudes (5000 ft). The synthetic aperture radar (SAR) has the ability to look through clouds. The SAR operates in spot and strip map mode. The strip map mode has a 30 cm resolution at an altitude of 12,500 ft and a slant range of 6.6 km. The width of the swath is 3,300 ft, allowing 13,000 nm^2 to be covered in a mission (Hewish 1995). The SAR subsystem will function autonomously by executing a series of preplanned mission commands loaded prior to operation. Sensors in development which could be carried by the predator at a later date include signal intelligence (SIGINT), foliage penetrating radars, miniature spectrometers, and gas chromatography to provide chemical analysis. Providing datalink for onboard sensors are a C-Band line of sight (LOS) and a UHF/Ku-band over the horizon (OTH) satellite data link. Also, an identification friend or foe (IFF) transponder is integrated into the onboard avionics package.

The predator is operable in mildly adverse weather, equivalent to instrumented flight by light civil aircraft. Icing, heavy precipitation, or high surface winds may prevent or affect launch operations.

The threat to the predator MAE at 15,000 feet include tactical surface to air missiles (SAM) and combat aircraft. The threat to the predator at 5,000 ft is much broader, including anti aircraft artillery (AAA) that could greatly increase the attrition of the predator. Also, the datalink pathways are susceptible to intercept or jamming since MAE employ LOS or UHF satellite communications (SATCOM) for command and control uplink and data downlink. Furthermore, current rules of engagement (ROE) call for the predator to operate outside known SAM engagement envelopes (Janes 1994).

Table 7. Tactical SAM's capabilities

Tactical SAM's	*Maximum effective range*	*Maximum effective altitude*
SA-4a	55,000m	27,000m
SA-4b	45,000m	24,000m
SA-6	24,000m	11,000m
SA-8a/b	12,000m	5,000m
SA-12a	15,000m	25,000m
SA-12b	100,000m	30,000m

II.3. Specific Problem

Air Combat Command wants to know the minimum number of predator UAVs required to meet a pre-specified expected coverage percent for a target set. The capabilities and the projected employment of the predator require a model to provide insight to the question above.

This research produces a heuristic model of the employment of the predator (MAE) UAV against a target set. The target set selected was from a fictitious country called QARI and NARI (threat density approximates that of a Middle Eastern country). The data set and target country for this research question is notional. The actual data is classified, so this thesis identifies a methodology and model for the user who deals with classified data. The random nature of the predator's attrition is the principal reason for modifying an existing TABU search algorithm developed by Carlton (1995).

III. Appendix 2

The problem of determining the number of vehicles or to service customers has been studied as a simulation, as a maximum covering problem (MCP), and vehicle routing problem (VRP).

III.1. Simulations

Simulations can provide insight into complex interrelationships over time, and therefore would be a tool of choice for predator operations. A previous simulation effort of remotely piloted vehicle (RPV) employs the SimScript II.5 language, and models the BGM-34C RPV drone launched from DC-130 aircraft during the Vietnam war (Mass 1976). Unfortunately, this previous simulation of UAVs provides little assistance for the development of this predator-based model due to the completely different profile used.

III.2. Maximum Covering Problem

A covering problem involves a set of items S and subsets of this set. For this problem the set S would include all the areas observed of dimension t (targets). Let there be n subsets of S such that $S \subset \bigcup_{j=1}^{n} S_j$. Denote the corresponding minimum distance tour of S_j as c_j and let $\mathbf{c}=(c_1,c_2,\ldots c_n)$. This cost c_j associated with subset S_j will be the minimum feasible time required for the predator UAV to perform surveillance of targets in S_j. This is a TSP problem in itself and is NP-hard (Lenstra 1981). Let $\mathbf{A}_{t\mathrm{x}n}$ matrix consist of column vectors a_j, where each element is 1 if target t is a member of subset j, or otherwise 0. Thus, vector a_j describes the set of targets a predator will cover. (A complete enumeration of a_j results in A_{txn} becoming prohibitively large very quickly.) For example, the vector $a_2=(10101)^T$ indicates that S_2 contains targets 1,3, and 5. The covering problem is to select a minimum cost collection of subsets so that set S is included in the union of collected subsets. The statement of the problem is:

$$\text{Min } c \cdot x$$

$$\text{Such that}$$

$$Ax \geq e$$

Where x is binary, and e is a unitary column vector of dimension t

While covering problems are one of the best studied problems in the field of integer programming, maximum covering problems (MCP) are often too complex to be solved in a reasonable amount of time. Since the MCP is NP-complete (Laporte 1992), the existence of an efficient (polynomial time) algorithm is unlikely, and solution times will increase exponentially with problem size. However, there are many heuristics which will give an approximate solution. A heuristic works by cleverly selecting variables to include in the cover set. The heuristic weights the candidate variables and selects the one that covers the most with the least cost. The candidate variables are reduced, and the heuristic iteratively develops a reasonable feasible solution. Gonsalvez *et al.* (1987) describes ten different heuristic algorithms for set covering.

III.3. General Vehicle Routing Problem

Vehicle routing problems is another way to look at the problem of number of predators to cover a target set. The general vehicle routing problem (GVRP) determines which routes allow a set of vehicles (predators) to service all customers (targets) at the minimum cost. Due to the complexity of GVRP they remain among one of the most difficult problems to solve. Despite a great deal of research only small problems, or problems of special structure, can be solved to optimality (Bodin 1983). Methods to solve the VRP include heuristics that give approximately optimal solutions.

The VRP consists of a vehicle fleet (UAVs) delivering products stored at central facility to satisfy customer orders for some period of time, where the fleet has fixed capabilities (speed, endurance etc.). Decisions are made to minimize the cost of operating the vehicle fleet. The routing decision involves determining which of the demands will be satisfied by each vehicle and what route each vehicle will follow in servicing demand.

The notation for the integer programming formulation of the VRP is:

NV=maximum number of available vehicles.

n=number of customers to which a delivery must be made.

K_v=unit capacity of vehicle.

T_v=maximum time allowed for any route of vehicle v.

d_i=unit demand at node i.

p^v_i=time required for vehicle v to deliver or collect at node i.

$t^v_{i,j}$=travel time for vehicle v from node i to node j.

$c_{i,j}$=cost of direct travel from customer i to j.

$x^v_{i,j}$=1, if vehicle v travels directly from customer i to customer j.
0, otherwise

The integer programming formulation of the problem of routing to minimize cost subject to vehicle capacity constraints is given below (Bodin 1983).

$$\text{MIN} \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{v=1}^{NV} c_{i,j} x^{v}_{i,j}$$

$$\sum_{i=1}^{n}\sum_{v=1}^{NV} x^{v}_{i,j} = 1 \quad j=2,....n \qquad \text{(one vehicle per node entering)}$$

$$\sum_{i=2}^{n}\sum_{v=1}^{NV} x^{v}_{i,j} = 1 \quad j=1,...n \qquad \text{(one vehicle per node exiting)}$$

$$\sum_{i=1}^{n} x^{v}_{i,i} = 0 \quad v=1,...NV \qquad \text{(prevents cycling)}$$

$$\sum_{i=1}^{n} x^{v}_{i,p} - \sum_{j=1}^{n} x^{v}_{p,j} = 0 \quad v=1,...NV{:}p=1,...n \text{ (every vehicle entering node must exit)}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} (d_i \cdot x^{v}_{i,j}) \le K_v \quad v=1,...NV \qquad \text{(vehicle capacity constraints)}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} p^{v}_{i} \cdot x^{v}_{i,j} + \sum_{i=1}^{n}\sum_{i=1}^{n} t^{v}_{i,j} \cdot x^{v}_{i,j} \le T_v \quad v=1,...NV \qquad \text{(total route elapsed time)}$$

Since the predator is not delivering or picking up anything, the capacity constraint of goods delivered can be relaxed. This makes the problem a traveling salesman problem (TSP). If UAVs are modified to deliver ordnance or supplies in the future, this constraint would be re-instated. A practical additional constraint to the GVRP is time window (TW) constraints. These constraints are required whenever service or surveillance of an area must be performed at certain time periods. Unfortunately, time windows give the GVRP much different characteristics. For instance, it is a trivial problem to find a feasible solution to the traveling salesman problem (TSP) since any ordering of customers is a feasible solution. However, Savelsbergh (1992) shows that even finding a feasible solution to the TSP with time windows is an NP-complete problem. Finally, there are heuristic approaches to time window constrained problems.

III.4. Heuristics

Tour construction algorithms build feasible routes starting with the depot and adding customers until no customers are left to be serviced. As soon as a feasible tour is found the algorithm is terminated. This procedure can be extended to the vehicle routing problem time window constrained (VRPTW), where several sequential and parallel procedures can construct feasible tours.

Also, there are several search algorithms applied to the VRP and VRPTW (Carlton 1995). Simulated annealing attempts to mimic the physical process that occurs when materials cool. This adaptive search procedure uses randomization to achieve final stability at a reasonable solution. The greedy randomized adaptive search procedure (GRASP) attempts to combine the best ideas from deterministic and randomized searches. GRASP searches the feasible region by randomly choosing among the best of neighboring solutions. Genetic algorithms attempt to use the theory of genetics in order to produce a good solution from a population of solutions. The genetic algorithm generates a population of solutions, from where the best of these generates another population. The final search algorithm discussed is the TABU method which attempts to avoid becoming stuck in local optima by exploiting memory and data structures to prevent returning to a previously examined solution.

III.5. TABU Search

The TABU search is a robust strategy for solving combinatorial optimizations like traveling salesman, graph coloring, job shop flow sequencing, integrated circuit design and time tabling problems. The strength of the TABU search is that it tracks the history of the search. The recency or frequency that certain moves have participated in past solutions are called attributes. Frequency of attributes, and particularly attributes of dominating solutions of local optima, give rise to strategies called intensification strategies.

Effective searches do not frequently re-visit a solution. To prevent the TABU search of revisiting swap moves tried recently, those moves are declared TABU. An example will pull this terminology together.

1	**2**	**3**	**4**	**5**

Figure 15. Original node arrangement (objective function value = 10)

1	**3**	**2**	**4**	**5**

Figure 16. Swapping nodes 2 and 3 (objective function value = 16)

In this example assume, the "swap" that maximizes the objective function value the most is reversing the positions of 2 and 3. Since the objective function value increases to 16, the move value is 16-10=6 (The move value of each swap represents the change in objective by the swap.) The nodes 2 and 3 are now TABU and will not be switched back unless either the aspiration criteria overrides the TABU status, or its TABU status expires.

In most cases the aspiration criteria is when the objective function value exceeds the best so far and the move is TABU. The actions of exploring neighborhood solutions are the intensification property of the heuristic. The actions of the TABU length force the heuristic into new areas (diversification).

There are N! permutations of possible arrangements of nodes. If we are concerned with the three center nodes there is 3! permutations. Therefore, a systematic method for "inserting" or swapping nodes is required (For the TABU heuristic nodes are inserted forward and backward. In the example above, node 3 was inserted before node 2.)

TABU search methods operate by defining a neighborhood from which adjacent solutions can be constructed that can be reached from the current solution. With this assumption, the reactive TABU search incorporates the basic parameters and memory structures of TABU search. Additionally, routines allowing the algorithm to automatically adjust search parameters based on the quality of search are used. For instance, the number of swap moves before the recurrence of a solution adjusts the TABU length. If the solution is visited within the defined cycle length, the TABU length is increased by the multiplicative factor; otherwise, the TABU length may be shortened.

Since the heuristic stops when the programmed number of iterations are accomplished, there is no guarantee of optimality. However, the TABU search has the "ability ...to obtain high quality solutions with modest computational effort, generally dominating alternative methods tested (Glover 1989)."

III.6. Conclusion

At first glance, using a formulation of the maximum covering problem or vehicle routing problem appears the best way to solve the problem of UAVs to cover a target set. However, the literature shows this problem belongs to the NP-complete class of models. Furthermore, an integer programming formulation requires a variant of stochastic programming, where the IP's would be sampled due to potential UAV losses. Consequently, this formulation would *not* lead to quick set up and execution times; therefore, this research is guided by the primary assumption that the best model where the processes are portrayed with adequate resolution is a heuristic. Since the TABU search heuristic best meets these requirements, it is used as the "engine" to develop a near-optimal solution to the multiple homogenous vehicle traveling salesman problem with time windows (mTSPTW).

IV. Appendix 3

IV.1. Validation

The original code was validated by comparing optimum 25-customer solutions. In two cases the TABU heuristic produces a solution greater than the optimum solution. However, in all cases tested below the TABU solution provides high quality solutions much faster than an integer programming solution.

Table 8. Revalidation of TABU heuristic

Problem Number	*TABU search Minimum travel time*	*Vehicles Used in TABU search*	*Iterations to best solution*	*Time to best*	*Optimum Minimum Travel time*	*Vehicles Used for optimum solution*	*Comp. Time for optimum solution*
c101	2441.3	3	14	.26	2441.3	3	18.6
c102	2484.6	3	225	.27	2440.3	3	79.9
c103	2440.3	3	138	.30	2440.3	3	134.7
c104	2476.3	3	38	.33	2436.9	3	223.9
c105	2441.3	3	39	.10	2441.3	3	25.6
c106	2441.3	3	14	.34	2441.3	3	20.7
c107	2441.3	3	149	.26	2441.3	3	31.7
c108	2441.3	3	210	.30	2441.3	3	43.1
c109	2441.3	3	210	.30	2441.3	3	585.4

Incorporating winds into the TABU heuristic merely changes vectors input into the heuristic. However, incorporating risk into the TABU heuristic does change how the algorithm searches by introducing the risk penalty into the objective function. In order to validate that this part of the code, the penalty was varied. Zero risk penalties give route structures identical to minimum route structures. On the other hand, high risk penalties tend to assign one vehicle to serve every target. This is the expected result since it minimizes the total risk for UAVs in the highest threat area.

The expected node coverage also changes the objective function in the TABU heuristic. Maximizing expected node coverage route heuristic gives the minimum route structure for low penalties. However, when advantageous the direction the predator travels will reverse, i.e. the vehicle will move in directions from high probability of survival to low probability of survival (assuming time window feasibility). High penalties for maximizing expected node coverage result in the minimum route structure being overridden, with both increased expected node coverage and additional predators.

V. APPENDIX 4

V.1. Computer Code

Computer code adapted from Carlton (1995) code to model winds

```
/* This program converts an input file of (x,y) integer coordinates to a
distance matrix ti(i,j) for an n node TSPTW, where n = number of nodes
including the depot the program outputs the matrix of INTEGERS ti(i,j)
to a designated time (distance) file. This routine is tailored to input
and compute the data
for the Solomon set of problems.  The data is scaled by a factor of 10
and
truncated.  */

#include <math.h>
/*#define PRINT_TIME 1          /*Uncomment to print time matrix.*/
#define FACTOR 10.0             /*Multiplies TWs and t(i,j) & s(i)'s to
increase
                           accuracy, yet use integer computations.*/

input_pbm(nc, nv, g, t, ti, ifp, ofp)
      int nc;                   /* The number of customers incl the depot.
*/
      int nv;                   /* The number of vehicles.*/
      int g;                    /* The penalty for add'l vehicle.*/
      int **ti;           /* The resulting time (distance) matrix. */
      struct node *t;           /* The node structure.*/
      FILE *ifp;          /*The pointer to the problem input file.*/
      FILE *ofp;          /*The pointer to the problem output file.*/

{
      float *x, *y;             /* The x, y coordinates of the customers
and
                            the depot.*/
      float *ea, *la;     /* The input vectors for the early and late TWs,
                           respectively.*/
      float *s;            /*Service time.*/
      int *d;                   /*Demand of customer i.*/

      int i, j;                 /* Index.*/

      x = (float *)calloc(nc, sizeof(float));
      y = (float *)calloc(nc, sizeof(float));
      ea = (float *)calloc(nc, sizeof(float));
      la = (float *)calloc(nc, sizeof(float));
      d = (int *)calloc(nc, sizeof(int));
      s = (float *)calloc(nc, sizeof(float));

 /*Input Depot and Customer data from the input file.*/

      for (i=0; i < nc; ++i)  {
            t[i].id = i;
            t[i].type = 1;
          fscanf(ifp,"%f",  &x[i]);
```

```
            fscanf(ifp,"%f", &y[i]);
            fscanf(ifp,"%d", &d[i]);
            fscanf(ifp,"%f", &ea[i]);
            fscanf(ifp,"%f", &la[i]);
            fscanf(ifp,"%f", &s[i]);
            t[i].e = (int) FACTOR*ea[i];
            t[i].l = (int) FACTOR*la[i];
            t[i].qty = d[i];
            }/*end for*/

 /*for (i=0; i < nc; ++i)  {

          fprintf(ofp,"x[%d] is %f",i,x[i]);
            fprintf(ofp,"y[%d] is %f",i,y[i]);
            fprintf(ofp,"%d", d[i]);
            fprintf(ofp,"%f", ea[i]);
            fprintf(ofp,"%f", la[i]);
            fprintf(ofp,"%f", s[i]);

            }*/
      /*end for*/

      t[0].type = 2;          /*Reset the 0-depot to the correct type.*/

/*Initialize the remaining vehicle nodes to be the same as the depot
node
            "Homogeneous" */

      for (i = nc; i < nc+nv; ++i) {
            t[i].id = i;
            t[i].e = t[0].e;
            t[i].l = t[0].l;
            t[i].type = 2;
            } /*end for*/

      free(ea);
      free(la);
      free(d);

      convxy(nc, nv, g, x, y, s, ti, ofp);

      free(x);
      free(y);
      free(s);

      return;

}/*end input_pbm function*/

convxy(nc, nv, gamma, x, y, s, ti, ofp, lengthx, lengthy)

      int nc;                 /* The number of customers incl the depot.
*/
      int nv;                 /* The number of vehicles.*/
      int gamma;        /* The penalty value for using an add'l veh.*/
      float *x, *y;    /* The x,y coordinates of nodes. */
      double lengthx;  /*The length of x vector*/
      double lengthy;  /*The length of y vector*/
      float *s;         /*Service time vector.*/
      int **ti;         /* The resulting time (distance) matrix. */
      FILE *ofp;          /* The pointer to the problem output file.*/

{
      int i,j, k;                /* Indices. */
      double dx, dy, dz;               /* The difference between the
respective x and y
                                coordinates.*/
      double power();            /* The power function prototype. */
      int nn;                    /* Computes nnodes.*/

      nn = nc + nv;

 /*****Do the conversion.******/

      for (i=0; i< nn; ++i){          /* Initialize the distance matrix. */
      for (j=0; j< nn; ++j)
            ti[i][j] = 0;}


      /*for (i=0; i< nc; ++i){
      fprintf(ofp,"\ni is %d",i);
      for (j=i+1; j< nc; ++j){
            dx = x[i] - x[j];
            dy = y[i] - y[j];
         dz=sqrt((dx)*(dx)+(dy)*(dy));
       lengthx=0.0*sqrt(dx*dx + dy*dy);
         lengthy=0.2*sqrt(dx*dx + dy*dy);
         fprintf(ofp,"\nlengthx is %5f",lengthx);
         fprintf(ofp,"\nlengthy is %5f",lengthy);


      fprintf(ofp,"\ny[%d] is %5f",i,y[i]);
         fprintf(ofp,"\ny[%d] is %5f",j,y[j]);

         if (y[i]==y[j]) {

         ti[i][j] = (int)(FACTOR*sqrt(dz*dz+lengthy*lengthy));
         ti[j][i] = ti[i][j];

         fprintf(ofp,"\nyeti[%d][%d] is %5d",i,j, ti[i][j]);
         fprintf(ofp,"\nyeti[%d][%d] is %5d",j,i, ti[j][i]);
              }

         else if (y[j]>y[i]) {
         ti[i][j] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-
 5+2*cos(atan(abs(dx/dy))))));
         ti[j][i] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-5+2*cos(180-
 atan(abs(dx/dy))))));
        fprintf(ofp,"\ngti[%d][%d] is %5d",i,j, ti[i][j]);
         fprintf(ofp,"\ngti[%d][%d] is %5d",j,i, ti[j][i]);

         }

         else {
         ti[i][j] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-5+2*cos(180-
 atan(abs(dx/dy))))));
         ti[j][i] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-
 5+2*cos(atan(abs(dx/dy))))));
         fprintf(ofp,"\nlti[%d][%d] is %5d",i,j, ti[i][j]);
         fprintf(ofp,"\nlti[%d][%d] is %5d",j,i, ti[j][i]);
```

```
          }


      }
      }*//*end for j*/
      for (i=0; i< nc; ++i){
      fprintf(ofp,"\ni is %d",i);    /* Computation. */
      for (j=i+1; j< nc; ++j){
            dx = x[i] - x[j];
            dy = y[i] - y[j];
         dz=sqrt((dx)*(dx)+(dy)*(dy));
       lengthx=0.1*sqrt(dx*dx + dy*dy);
         lengthy=0.0*sqrt(dx*dx + dy*dy);
         fprintf(ofp,"\nlengthx is %5f",lengthx);
         fprintf(ofp,"\nlengthy is %5f",lengthy);

         /*if (y[j]>=y[i]) {*/
      fprintf(ofp,"\nx[%d] is %5f",i,x[i]);
         fprintf(ofp,"\nx[%d] is %5f",j,x[j]);

         if (x[i]==x[j]) {

         ti[i][j] = (int)(FACTOR*sqrt(dz*dz+lengthx*lengthx));
         ti[j][i] = ti[i][j];

         fprintf(ofp,"\nxeti[%d][%d] is %5d",i,j, ti[i][j]);
         fprintf(ofp,"\nxeti[%d][%d] is %5d",j,i, ti[j][i]);
               }

         else if (x[j]>x[i]) {
         ti[i][j] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-
5+2*cos(atan(abs(dy/dx))))));
         ti[j][i] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-5+2*cos(180-
atan(abs(dy/dx))))));
       fprintf(ofp,"\ngti[%d][%d] is %5d",i,j, ti[i][j]);
         fprintf(ofp,"\ngti[%d][%d] is %5d",j,i, ti[j][i]);

         }

         else {
         ti[i][j] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-5+2*cos(180-
atan(abs(dy/dx))))));
         ti[j][i] = (int)(FACTOR*.0465847*sqrt(-96*dz*dz*(-
5+2*cos(atan(abs(dy/dx))))));
         fprintf(ofp,"\nlti[%d][%d] is %5d",i,j, ti[i][j]);
         fprintf(ofp,"\nlti[%d][%d] is %5d",j,i, ti[j][i]);
         }



      }
      }/*end for j*/


/*Process to ensure that the triangle inequality holds:*/

      for(i=0; i < nc; ++i)
         for(j=i+1; j < nc; ++j)
            for(k=0; k < nc; ++k) {
```

```
            if( k==i || k==j ) continue;
            if(ti[i][j] > ti[i][k] + ti[k][j])
               ti[j][i] = ti[i][j] = ti[i][k] + ti[k][j];
          }/*end for*/

      for (i=1; i< nc; ++i)
         for (j = nc; j < nc+nv; ++j)
            ti[i][j] = ti[j][i] = ti[i][0];

/* Add scaled service time to each distance vector.*/

      for (i=0; i< nc; ++i)
            for(j=0; j<nc+nv; ++j)
                  ti[i][j] += (int) FACTOR*s[i];
                              /*divide s[i] by two for "half s"*/

      for (i=nc; i< nc+nv; ++i)
            for(j=0; j< nc+nv; ++j)
                  ti[i][j] += (int) FACTOR*s[0];
                              /*divide s[0] by two for "half s"*/

/*Complete the matrix by input the variable vehicle usage "cost."*/

      for (i = nc; i < nn-1; ++i)
            for (j= i+1; j < nn-1; ++j)
                  ti[i][j] = ti[j][i] = gamma;

      for (i = nc; i < nn-1; ++i)
            ti[i][0] = ti[0][i] =ti[i][nn-1] = ti[nn-1][i] = gamma;

/*****Produce the output.************/
#     if PRINT_TIME
      fprintf(ofp, "The time factor for scaling is %5.1f \n", FACTOR);
      fprintf(ofp, "%d", nn);
      for (i=0; i< nn; ++i)  {
            fprintf(ofp, "\n");
      for (j=0; j< nn; ++j)
            fprintf(ofp, "%5d", ti[i][j]);
      } /*end for i*/

      fprintf(ofp, "\n");
#endif

      return;

} /*******The end of the convxy
routine.************************************/
```

C++ computer code subroutine to incorporate minimum risk

```
/*************Computes route penalty*********/
comp_routepen(n,rpen,t,k,ofp)
        int n;                          /* The number of nodes in tour.*/
        float *rpen;                    /* The tour penalty*/
        struct node *t; /*The current tour*/
        int k;
```

```
{

          float routepen;                    /*The total penalty of the tour.*/
          float y;
          float z;
          float q;
          float bb;
          float zz;/*dummy variable*/
   int i;
          int j;/*counter*/
   int x;
          int aa;
          float *rp;
   /*rp = (float *)calloc(n, sizeof(float));*/
          routepen = 0.0;
          z=0.0;
          y=1.0;
          q=1.0;
          bb=0.0;
          zz=0;
          x=0;
          aa=1;



                  for (i = 1; i <= n-1; ++i)  {
                  if(t[i].type == 1) {
                  x=t[i].id;

                    y=y*rpen[x];



                  }
                  else {

                          q=min(q,y);

                          routepen=(int)(100000.0*(1-q));

                  }

          } /*end for*/
          /*fprintf(ofp,"\nroutepenin is %f\n",routepen);*/

          /*free(rp);*/
          return routepen;


 }                    /*This ends the comp_routepen function.*/
```

C++ computer code subroutine to incorporate maximum coverage

```
/**************Computes route penalty**********/
comp_routepen(n,rpen,t,k,ofp)
            int n;                          /* The number of nodes in tour.*/
            float *rpen;                    /* The tour penalty*/
            struct node *t; /*The current tour*/
            int k;

{

            float routepen;                 /*The total penalty of the tour.*/
            float y;
            float z;
            float q;
            float bb;
            float zz;/*dummy variable*/
    int i;
            int j;/*counter*/
    int x;
            int aa;
            float *rp;
    rp = (float *)calloc(n, sizeof(float));
            routepen = 0.0;
            z=0.0;
            y=0.0;
            q=1.0;
            bb=0.0;
            zz=0;
            x=0;
            aa=1;

            /*for (i = 1; i <= n-1; ++aa)  {
                        rp[i]=0;
            }*/



            for (i = 1; i <= n-1; ++i)  {
                        if(t[i].type == 1) {
                        x=t[i].id;
    /*printf("\nnode number is %d\n",x);*/
                         /*y=y*rpen[x];*/

            /*z=rpen[aa-1]+rpen[aa-1]*rpen[aa];*/
                        rp[aa]=rpen[x];


                                    /*fprintf(ofp,"\nrp[aa] is %f\n",rp[aa]);
                                    fprintf(ofp,"\nrp[x] is %f\n",rp[x]);*/
                                    aa+=1;


                                    /*printf("\npenalty for node is %f\n",rpen[x]);*/

                        }
```

```
        else {

                /*q=min(q,y);

                routepen=(int)(1000.0*(1-q));*/
                /*fprintf(ofp,"routepen is %lf",routepen);*/
        /*fprintf(ofp,"rp[%d]is %f",aa,rp[aa]);         */
                        /* y=1.0;*/
    /*zz=(zz+y);*/


    z=z+(rp[1]+rp[1]*rp[2]+rp[1]*rp[2]*rp[3]+rp[1]*rp[2]*rp[3]*rp[4]+rp[1]*rp[2]*rp[3]*rp[4]*rp[
5]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]+rp[1]*rp[2]*rp[3]*rp
[4]*rp[5]*rp[6]*rp[7]*rp[8]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]+rp[1]*rp[2]*rp[3]*r
p[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10
]*rp[11]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]+rp[1]*rp[2]*rp[3]
*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]
*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8
]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp
[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]
*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*r
p[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]+rp[1]*rp[2]*rp[3]*r
p[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp
[19]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]
*rp[16]*rp[17]*rp[18]*rp[19]*rp[20]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[
11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*rp[20]*rp[21]+rp[1]*rp[2]*rp[3]*rp[4]*r
p[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*r
p[20]*rp[21]*rp[22]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13
]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*rp[20]*rp[21]*rp[22]*rp[23]+rp[1]*rp[2]*rp[3]*rp[4]*rp[
5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*rp[
20]*rp[21]*rp[22]*rp[23]*rp[24]);

    /*fprintf(ofp,"\nz is %f\n",z);*/
  routepen=1000*(1/z);

    aa=1;
    for (j = 1; j <= n-1; ++j)  {
            rp[j]=0;
    }

    /*printf("The iteration is %d",k);*/

    }

            }
    } /*end for*/
    /*fprintf(ofp,"\nroutepenin is %f\n",routepen);*/

    free(rp);
    return routepen;


}                   /*This ends the comp_routepen function.*/
```

/*************A function to perform a swap move.***************************/

The following is Carlton's code with the subroutine call comp_routepen in bold.

```
/* This code performs the input and constructs an initial tour for an mtsptw;
 This program incorporates the incremental cost of a neighboring solution.

 The code incorporates the STRONG TW checks.  The code uses a hashing structure
 that identifies whenever a solution is repeated, and then adjusts the search
 parameters appropriately.  The code also incorporates a time window reduction
 scheme and a procedure to compute the amount of TW overlap before and after TW
 reduction This algorithm computes the time to the best solution found as well
 as the total computation time, the time to best solution found is output.

 This incorporates the heirarchical time penalty for saving the best infeasible
 tour found.

 This code uses redundancy between homogeneous vehicles & strong TW feasibility
 to limit the candidate list of neighbors without unduly restricting the
 search.*/

/*Version 1 reads in the Solomon problem set, converts the data to one decimal place, truncated, and
procedes as p1v7 applied to the UMontreal data set.*/

#include <time.h>
#include <math.h>

#define HTSIZE 1009                    /*The dimension of the hashing table*/

/******************REACTIVE SEARCH PARAMETER INITIALIZATIONS****************/

#define INCREASE  1.2
                /*Should guarantee to increase tabu length by at least one.*/

#define DECREASE  .9
                /*Allows for any function to be inserted.*/

#define CYMAX  50
                /*Allows for any function to be inserted.*/
/****************************************************************************/
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
#define max(a, b) ((a>b) ? (a) : (b))
#define min(a, b) ((a<b) ? (a) : (b))

/*#define PRINT_ITERS 1                  /*Uncomment to print the move value and other
                                    information.*/
/*#define PRINT_HASH 1                   /*Uncomment to print the hashing information.*/
/*#define PRINT_HTBL 1                   /*Uncomment to print the hashing table.*/

/************************STRUCTURES*************************************/

 typedef struct node {        /* The data structure for a vehicle or "task" node.*/
```

```
                int id;                         /*The node number.*/
                int e;                          /*The early time window.*/
                int l;                          /*The late time window.*/
                int qty;                /*The node demand or veh capacity.*/
                int type;               /*The node type: 1 => destination,
                                                 2 => vehicle.*/
                int arr;                /*The arrival time at the node.*/
                int dep;                /*The departure time from the node.*/
                int wait;               /*The waiting time at the node.*/
        } NODE;

/*The hashing structure.*/

        struct hashlist {
                unsigned thval;                 /*The tour hashing value.*/
                int cost;               /*The cost of the tour.*/
                int tvltime;                    /*The tour travel time.*/
                int pen;                /*The tour penalty value.*/
                int lastfound;                  /*The iteration on which the tour was
                                                 last visited.*/
                struct hashlist *next;          /*Pointer to the next item in the
                                                 list.*/
        } HASHLIST;


struct hashlist *hashtbl[HTSIZE];
                                /*The Hashtbl is a vector of pointers to hashlist
                                 structures.*/
#include "twred.h"
#include "solconv.h"
#include "printout.h"
#include "retainc.h"
#include "hash3.h"
#include "makespan.h"
#include "dlaout.h"

float *t_search(ifp, ofp, niters, PEN, numveh, g)
        int     niters;                 /* The number of iterations that the TS
                                         algorithm is allowed to run.*/
        FILE    *ifp;                   /*The pointer to the problem input file.*/
        FILE    *ofp;                   /*The pointer to the problem output file.*/
        double  PEN;                    /*The penalty for TW infeasibility.*/
        int     numveh;                 /*The number of vehicles in the problem, or 0.*/
        int     g;              /*The penalty for add'l vehicle.*/
{

#include "vars6.h"
int r_pennew;
int r_penold;
int hold;
int r_peninit;
/****************************INPUT*****************************************/
```

```
          k = numfeas = iter_no = bfiter_no = numdiff = 0;
          tour_short = 999999;
          bftour_cost = bftvl_time = feas_compl = 999999;
          num_veh_used = num_feas_veh = 0;

/* Input the data from a data file*/
      fscanf(ifp,"%d", &numcust);  /*Input the number of customers from the
                                             input file, including the depot.*/
          ++numcust;                         /*Increases numcust to incl the depot.*/

/*Determine the number of vehicles to be modelled.  If no vehicles are input, or
 number of vehicles is more than the number of customers then model one vehicle
 for each customer.*/

          if (numveh == 0 || numveh >= numcust)  numveh = numcust-1;

          nnodes = numcust + numveh;  /*The total number of modelled nodes.*/

/*Lines to dynamically allocate memory for the problem vectors and matrices based on the number of
nodes actually in the problem.  */

          m =  (int *)calloc(nnodes, sizeof(int));
          z = (int *)calloc(nnodes, sizeof(int));

          tour = (struct node *)calloc(nnodes, sizeof(NODE));
     best_tour = (struct node *)calloc(nnodes, sizeof(NODE));
    best_ftour = (struct node *)calloc(nnodes, sizeof(NODE));
       oldtour = (struct node *)calloc(nnodes, sizeof(NODE));

          time = (int **)calloc(nnodes, sizeof(int *));
rpen = (float *)calloc(numcust, sizeof(float));
          for (i=0; i < nnodes; ++i)
                  time[i] = (int *)calloc(nnodes, sizeof(int));


          tabu_list = (int **)calloc(nnodes, sizeof(int *));
          for (i=0; i < nnodes; ++i)
                  tabu_list[i] = (int *)calloc(nnodes-1, sizeof(int));

          input_pbm(numcust, numveh, g, tour, time,rpen, ifp, ofp);

/*Input the number of iterations, tabu length, and depth of search.  */

          mavg = nnodes-2;
          ssltlch = 0;
          tabu_length = min(30, nnodes-2);
          depth =  nnodes-2;

/*Start Timing Here!*/

          start = clock();

/*Initialize the vector z[i] of random numbers.*/
```

```
        srand(1);

        for (i=0; i<numcust; ++i)
                z[i] = 1 + (int) (131072.0*rand()/(RAND_MAX+1.0));
        for (i = numcust; i < nnodes; ++i) z[i] = z[0];

#       ifdef PRINT_HASH
        fprintf(ofp,"\n");
        for(i=0; i<nnodes; ++i)
                fprintf(ofp,"Z[%d] = %d\n", i, z[i]);
#       endif

/*Initialize the hashing table to all NULL pointers.*/
        for (i = 0; i<HTSIZE; ++i)
                hashtbl[i] = NULL;

/*Conduct the Tw Reduction.*/

        twredol(nnodes, tour, time, ofp);

/*********************DETERMINE INITIAL TOUR**********************************/

/*A.  This code produces a tour based on a sort of increasing average time windows at each node */

/*1.  Compute the average time window at each node "m[i]," excluding the depot nodes. */

        for (i=1; i < nnodes; ++i)
                m[i] = (tour[i].e + tour[i].l)/2;

/* 2.  Sort (bubble sort) the initial tour based on the avg TW time. Also do not swap if the customers do
not satisfy strong TW feasibility.*/

        for (i = 1; i < numcust-1; ++i)
        for (j = numcust-1; j > i; --j)
         if (m[j-1]> m[j] &&
             tour[j].e + time[tour[j].id][tour[j-1].id] <= tour[j-1].l) {
                SWAP(m[j-1], m[j], temp1);
                SWAP(tour[j-1], tour[j], temp2);
         }/* end if*/

        print_tour(tour, nnodes, ofp);      /* Print the initial tour.*/

/*B.  This computes the initial schedule for the initial tour, and stores the values in the node structure
returns the total tour length excluding any penalty for infeasibility.*/

        tour[0].arr = tour[0].e;
        tour[0].dep = tour[0].e;  /*Initialize starting values at the depot.*/
        tour[0].wait = 0;    /*These will never change for this model.*/

        tour_length = tour_sched(1, nnodes, tour, time);

/*C.  This  computes the initial cost of the tour = tour_length + penalty for infeasibility.*/

        time_penalty = comp_timepen(nnodes, tour);
```

```
  fprintf(ofp,"timepenalty is %d",time_penalty);
  r_peninit= (comp_routepen(nnodes,rpen,tour,k,ofp));
  time_penalty += r_peninit;
        fprintf(ofp,"timepenalty and rpeninit is %d",time_penalty);


        if (time_penalty > 0)  {
                fprintf(ofp,"This starting tour is infeasible!\n");
                bt_pen_cost = time_penalty; } /*Initialize the parameter.*/
        else bt_pen_cost = 9999;

        pen_cost = PEN*time_penalty;

        tour_cost = tour_length + pen_cost;

        totwait = sum_wait(nnodes, tour);
        best_cost = tour_tt = tour_cost - totwait;
        tvl = best_tt = tour_tt - pen_cost;
        compl_time = tour_length;

        best_time = 0.0;
        bestf_time = 9999.0;

        for (i= 0; i< nnodes; ++i)
                best_tour[i] = tour[i];

        fprintf(ofp,"The tour cost is %9.1f, the travel time is %9.1f.\n", (float)tour_cost/FACTOR, (float)
best_tt/FACTOR);

/* Compute the hashing value for the initial tour.*/

        h3t = 0;
        for (i = 0; i < nnodes-1; ++i)
                h3t += z[tour[i].id]*z[tour[i+1].id];

        tourhv = h3t;

#       ifdef PRINT_HASH
        fprintf(ofp,"h3 = %u.\n", h3t);
#       endif
/*****************************************************************************/
/*                    TABU SEARCH SUBROUTINE                          */
/*     EVALUATES ALL INSERT NEIGHBORS AND FINDS THE BEST TO CHANGE          */
/*                 OUTPUTS NEW TOUR AND SCHEDULE                         */
/*                    AT EVERY ITERATION                         */
/*****************************************************************************/

        for (i = 0; i< nnodes-1; ++i)
         for (j = 0; j< nnodes-1; ++j)
                tabu_list[i][j] = 0;            /* Initialize tabu structure. */

        fprintf(ofp,"\nTabu_length is %d and the Number of iterations is %d. \n", tabu_length, niters);
        fprintf(ofp,"The depth of the search is %d.\n", depth);
        ++k;      /* Increment k to k = 1.*/

        if (time_penalty == 0)
          keep_bfs(nnodes, tour, tour_cost, tvl, best_ftour, &bftour_cost, &bftvl_time, &bfiter_no, k-1,
start, &bestf_time);

 while (k <= niters) {

#       if (PRINT_ITERS || PRINT_HASH)
fprintf(ofp,"*****************ITERATION NUMBER %3d
************************************\n\n", k);
        if (time_penalty == 0)
        fprintf(ofp,"      *************This tour is feasible!***************\n");
        print_tour(tour, nnodes, ofp);   /*To print the newest incumbent tour.*/
        fprintf(ofp,"The tour cost is %d.\n", tour_cost);
#       endif

/*This is the effort to determine if any tours are repeated and to adjust
the tabu_length accordingly.*/

        ptr = lookfor(tour_cost, tourhv, time_penalty, k-1, tvl);

        if (time_penalty == 0 &&  ptr == NULL)
                ++numfeas;

        if (ptr == NULL) {
                notfound(&tabu_length, &ssltlch, mavg, ofp);
                ++numdiff;
                }
        else
                found(ptr, &tabu_length, &ssltlch, &mavg, k, ofp);

#       ifdef PRINT_HASH
        if (ssltlch == 0)  {
                fprintf(ofp,"The new tabu_length is %d.\n", tabu_length);
                fprintf(ofp,"The moving average is %5.2f.\n\n", mavg);
                }
#       endif

        d_best = esc_best = d_bestf = 99999;

        ch_i = feas_i = esc_i = 0;

 /**************** Check all "later" insertions.****************************/

        for (i=1; i< nnodes-2; ++i) {

                if(tour[i].type == 2) continue;
                                              /*Don't consider vehicle nodes.*/

          for (j=0; j < nnodes ; ++j)
            oldtour[j] = tour[j];
```

```
                    for (d = 1; d <= depth; ++d) {

                     if (i+d < nnodes-1) {

                       if(tour[i+d].e + time[tour[i+d].id][tour[i].id]> tour[i].l)
                              {  while (oldtour[i+d].type == 1)

                               {
/**If Str TWs violated within a vehicle, move the customer the tour along until a vehicle is
encountered...swap and "locally update" the schedule as the customer is swapped, and increment d
also.**/

                    swap_node(i+d-1, i+d, oldtour);
                    oldtour[i+d-1].arr = oldtour[i+d-2].dep +
                                  time[oldtour[i+d-2].id][oldtour[i+d-1].id];
                    oldtour[i+d-1].dep = max(oldtour[i+d-1].e, oldtour[i+d-1].arr);
                    oldtour[i+d-1].wait = oldtour[i+d-1].dep - oldtour[i+d-1].arr;
                              ++d; } /*end while customer type*/

                              if (i+d == nnodes-1) break;
                    /*If you increment to the last position in the tour, don't swap
with the end depot node!*/
                               }/*end if StrTW violated.*/

                      swap_node(i+d-1, i+d, oldtour);
          r_pennew= (comp_routepen(nnodes,rpen,tour,k,ofp));
          r_penold= (comp_routepen(nnodes,rpen,oldtour,k,ofp));
           move_val = move_delta(i, d, nnodes, tour, oldtour, time);
          hold=r_penold-r_pennew;
                               /*fprintf(ofp,"hold=%d",hold);
          fprintf(ofp,"move_val before=%d",move_val);*/
                               move_val += PEN*(r_penold-r_pennew);
                              /* fprintf(ofp,"move_val after=%d",move_val);
          print_tour(tour, nnodes, ofp);
                               print_tour(oldtour,nnodes,ofp);
          fprintf(ofp,"r_pennew=%d r_penold=%d",r_pennew,r_penold);*/




                               /*fprintf(ofp,"move_val+r_pen=%d",move_val);*/
                      nbrpen = comp_timepen(nnodes, oldtour)+r_penold;
                      move_val += PEN*(nbrpen - time_penalty);
                               /*fprintf(ofp,"move_val+time penalty=%d",move_val);*/



#         ifdef PRINT_ITERS
fprintf(ofp,"The move value and tour cost [%d, %d]: %d %d\n", tour[i].id, d, move_val, move_val +
tour_tt);
#         endif

          if (nbrpen == 0)  {
```

```
          if (move_val < d_bestf)  {
           if ( k > tabu_list[tour[i].id][i+d] || (move_val+tour_tt <
                                                                  best_cost)) {
                  d_bestf = move_val;
                  feas_i = i;
                  feas_d = d;
                   } /* end if not tabu*/
             }/*end if improved move value (travel time).*/
           } /*end if feasible neighbor found.*/

          else {
           if (move_val < d_best)  {
           if ( k > tabu_list[tour[i].id][i+d] || (move_val+tour_tt <
                                                                  best_cost)) {
                  d_best = move_val;
                  ch_i = i;
                  ch_d = d;
                   } /* end if not tabu*/
             } /*end if improved move value*/
           }/*end else: infeasible neighbor.*/

/*escape routine*/

          if (move_val < esc_best)  {/*Finds the best neighboring move.*/
                  esc_best = move_val;
                  esc_i = i;
                  esc_d = d;
                  }/* end escape if*/

             } /* end if feasible depth*/

                  if(oldtour[i+d+1].type == 2 && oldtour[i+d+1].id == i+d+1)
                        break;
                        /*If only vehicle nodes are left in the tour stop!*/

                  } /* end for d*/

          } /*end for i*/

/***************** Check all earlier insertions.****************************/

          for (i=3; i<=nnodes-2; ++i) {

                  if(tour[i].type == 2) continue;
                                             /*Don't consider vehicle nodes.*/

                  for (j=0; j < nnodes ; ++j)
                 oldtour[j] = tour[j];

                  d = 1;

                   if (tour[i].e + time[tour[i].id][tour[i-1].id] <= tour[i-1].l)
                        {   swap_node(i-d, i-d+1, oldtour);
                        ++d;  }/*end if no TW Violation*/
            else  {
                    while (oldtour[i-d].type == 1) {
                            swap_node(i-d, i-d+1, oldtour);
                            ++d; } /*end while customer is adjacent*/
                    if (i-d == 0)  continue;
                    }/*end else  Str TWs violated.*/


           for (d; d <= depth; ++d) {

            if (i-d > 0) {

              if(tour[i].e + time[tour[i].id][tour[i-d].id] > tour[i-d].l)
                    {   while (oldtour[i-d].type == 1) {
                            swap_node(i-d, i-d+1, oldtour);
                            ++d; } /*end while customer is adjacent*/
                    if (i-d == 0)  break;
                    }/*end if  Str TWs violated.*/

              swap_node(i-d, i-d+1, oldtour);
       r_pennew= (comp_routepen(nnodes,rpen,tour,k,ofp));
              r_penold= (comp_routepen(nnodes,rpen,oldtour,k,ofp));
              move_val = move_delta(i, -d, nnodes, tour, oldtour, time);
                hold=r_penold-r_pennew;
                    /*fprintf(ofp,"hold=%d",hold);
                     fprintf(ofp,"move_val before=%d",move_val);*/

                     move_val += PEN*(r_penold-r_pennew);
                    /* fprintf(ofp,"move_val after=%d",move_val);
  fprintf(ofp,"PEN=%d",PEN);
              print_tour(tour, nnodes, ofp);
              print_tour(oldtour,nnodes,ofp);
  fprintf(ofp,"r_pennew=%d r_penold=%d",r_pennew,r_penold);

              print_tour(tour, nnodes, ofp);*/
/*fprintf(ofp,"r_pen=%d",r_pen);*/

              /*fprintf(ofp,"move_val early=%d",move_val);*/
              /*move_val +=PEN* r_pen;*/
              /*fprintf(ofp,"move_val+r_pen=%d",move_val);*/
              nbrpen = comp_timepen(nnodes, oldtour)+r_penold;
              move_val += PEN*(nbrpen-time_penalty);
/*fprintf(ofp,"move_val+time penalty=%d",move_val);*/
#          ifdef PRINT_ITERS
fprintf(ofp,"The move value and tour cost [%d, %d]: %d %d\n", tour[i].id, -d, move_val, move_val +
tour_tt);
#          endif

          if (nbrpen == 0)  {

           if (move_val < d_bestf)  {
             if ( k > tabu_list[tour[i].id][i-d] || (move_val+tour_tt <
                                                                    best_cost)) {
                    d_bestf = move_val;
                    feas_i = i;
                    feas_d = -d;
                     } /* end if not tabu*/
               }/*end if improved move value (travel time).*/
             } /*end if feasible neighbor found.*/

          else {

           if (move_val < d_best)  {

             if ( k > tabu_list[tour[i].id][i-d] || (move_val+tour_tt<
                                                                    best_cost)) {
                    d_best = move_val;
                    ch_i = i;
                    ch_d = -d;
                     } /* end if tabu*/
               } /*end if improved move value*/
             }/*end else: infeasible neighbor.*/

/*escape routine*/

          if (move_val < esc_best)  {/*Finds the best neighboring move.*/
                    esc_best = move_val;
                    esc_i = i;
                    esc_d = -d;
                    }/* end escape if*/

               } /* end if feasible depth*/

                    } /* end for d*/

          } /*end for i*/

/*If a feasible move is found...move to it!*/
          if (feas_i != 0) {
                    ch_i = feas_i;
                    ch_d = feas_d;
             }/*end if feasible move is found.*/

/***************IF ALL MOVES ARE TABU AND NONE MEET
ASPIRATION*****************/
/*                                                                    */
/*          THEN SET CH_I AND CH_D TO THE BEST MOVE DISCOVERED            */
/*                    AND DECREASE THE TABU LENGTH                        */
/*            OR IF THERE ARE NO MOVES AVAILABLE                          */
/**************************************************************************/

          if (esc_i == 0) {
                    fprintf(ofp,"There are no moves available. . . \n\t Increase the number of vehicles
available and try again!\n");
                    printf(":  There are no moves available. . . \n\t Increase the number of vehicles available
and try again!\n");
```

```
                break; }/*end if no moves available.*/

        if(ch_i == 0)  {

#       ifdef PRINT_ITERS
                fprintf(ofp,"All moves at iteration %d are tabu and none meet the aspiration criteria.\n",
k);
#       endif

                ch_i = esc_i;
                ch_d = esc_d;
                tabu_length = max((int) (tabu_length)*DECREASE,5);
                }/* end for all moves tabu*/

/***************UPDATE: TABU LIST AND TOUR POSITIONS.**************************/

/*Allows no "return" moves for tabu_length iterations. */

        if (ch_d == 1)
          tabu_list [tour[ch_i+1].id][ch_i+1] = k + tabu_length;
        else
          tabu_list [tour[ch_i].id][ch_i] = k + tabu_length;

 /* Allows no "repeat" moves for tabu_length iterations. */

         tabu_list [tour[ch_i].id][ch_i+ch_d] = k + tabu_length;

#       ifdef PRINT_ITERS
fprintf(ofp,"\nThe move INSERTS Node %d, to position %d.\n\n", tour[ch_i].id, ch_i + ch_d);
#       endif

/*BEFORE the new tour is constructed, update the h3t value:*/

        zin = zout = 0;

        i= ch_i;
        j=((ch_d>0) ? ch_i+ch_d : ch_i+ch_d-1);

        zout = z[tour[i-1].id]*z[tour[i].id]
                +z[tour[i].id]*z[tour[i+1].id]
                  +z[tour[j].id]*z[tour[j+1].id];

        zin  = z[tour[i-1].id]*z[tour[i+1].id]
                +z[tour[j].id]*z[tour[i].id]
                  +z[tour[i].id]*z[tour[j+1].id];

        h3t += zin - zout;

        tourhv = h3t;

 #      ifdef PRINT_HASH
        fprintf(ofp,"\n");
        fprintf(ofp,"       Tour hashing value = %u.\n", tourhv);
 #      endif
```

```
          tour = insert(ch_i, tour, ch_d, nnodes);

/***************UPDATE: THE NEW INCUMBENT SCHEDULE.*****************************/

          tour_length = ((ch_d >0) ? tour_sched(ch_i, nnodes, tour, time)
                                     : tour_sched(ch_i+ch_d, nnodes, tour, time));

          time_penalty = comp_timepen(nnodes, tour);
   r_peninit= (comp_routepen(nnodes,rpen,tour,k,ofp));
          time_penalty += r_peninit;
          pen_cost =  PEN * time_penalty;
          /*fprintf(ofp,"pen_cost for incumbant schedule is %d",pen_cost);*/
          tour_cost = tour_length + pen_cost;
   /*fprintf(ofp,"tour_cost for incumbant schedule is %d",tour_cost);*/
          totwait = sum_wait(nnodes, tour);
          tour_tt = tour_cost - totwait;
          tvl = tour_tt-pen_cost;

          /*dlaout(nnodes, tour, time, 0.0, (float) tvl/FACTOR);*/

          if (time_penalty == 0)
           keep_bfs(nnodes, tour, tour_cost, tvl, best_ftour, &bftour_cost, &bftvl_time, &bfiter_no, k,
start, &bestf_time);

          if (tour_tt < best_cost) {
                   compl_time = tour_length;
                   best_tt = tvl;
                   bt_pen_cost = time_penalty;
                   best_cost = tour_tt;

/*This is a test routine.
          printf("\n %d %d %d %d %d", k, compl_time, best_tt, bt_pen_cost, best_cost);*/
                   soln_time = clock();
                   best_time = ((soln_time - start)/(double) CLOCKS_PER_SEC);

                   for (i=1; i< nnodes; ++i)
                     best_tour[i] = tour[i];
                   iter_no = k;
          } /* end if -- end the update of the best tour value & best tour.*/

  ++k;

  } /*end while....Ends the tabu search subroutine*/

/*Add the tour found at the last iteration to the hash table, if necessary.*/

          ptr = lookfor(tour_cost, tourhv, time_penalty, k-1, tvl);

          if (time_penalty == 0 &&  ptr == NULL)
                   ++numfeas;

          if (ptr == NULL)
```

```
                notfound(&tabu_length, &ssltlch, mavg, ofp);
        else
                found(ptr, &tabu_length, &ssltlch, &mavg, k, ofp);

        stop = clock();
        duration =  ((stop - start)/(double) CLOCKS_PER_SEC);

        if (bestf_time != 9999.0)
                makespan(nnodes, best_ftour, &feas_compl, &num_feas_veh);

        makespan(nnodes, best_tour, &compl_time, &num_veh_used);

/********** OUTPUT:
        BEST TOUR FOUND, ITERATION NUMBER, TOUR LENGTH, THE SHORTEST
         TOUR FOUND OVERALL REGARDLESS OF TW FEASIBILITY, AND THE NUMBER OF
TW
         FEASIBLE TOURS DISCOVERED DURING THE SEARCH.************************/

/*Record the solution values for the summary sheet.*/
        soln[0] =
                ((bestf_time != 9999.0)? feas_compl: compl_time)/FACTOR;
        soln[1] = ((bestf_time != 9999.0)? bftvl_time: best_tt)/FACTOR;
        soln[2] = bfiter_no;
        soln[3] = best_tt/FACTOR;
        soln[4] = ((bftvl_time == best_tt)? 0: bt_pen_cost)/FACTOR;
        soln[5] =  iter_no;
        soln[6] = ((bestf_time != 9999.0)? bestf_time: best_time);
        soln[7] = numfeas;
        soln[8] = compl_time/FACTOR;
        soln[9] = duration;
        soln[10] = ((bestf_time != 9999.0)? bfiter_no: iter_no);
        soln[11] = feas_compl/FACTOR;
        soln[12] = num_feas_veh;
        soln[13] = num_veh_used;
        soln[14] = 0;


        if (bftour_cost < 999999)  {
                fprintf(ofp,"The best feasible tour was found on the %dth iteration.\n", bfiter_no);
                print_sched(best_ftour, nnodes, ofp);
                fprintf(ofp,"The cost of the tour is %d.\n", bftour_cost);
                fprintf(ofp,"The travel time of the tour is %d.\n", bftvl_time);

                /*dlaout(nnodes, best_ftour, time, soln[0], soln[1]);*/

                }
        else
                fprintf(ofp,"THE ALGORITHM FOUND NO FEASIBLE TOUR.");

        if (best_tt != 0 /*bftvl_time*/)        {
                fprintf(ofp,"\n\nThe best overall travel time tour was found on the %dth iteration.\n",
 iter_no);
                print_sched(best_tour, nnodes, ofp);
                fprintf(ofp,"The length of the tour is %8.1f.\n", (float) compl_time/FACTOR);
```

```
            fprintf(ofp,"\nThe travel time of this tour is%8.1f.\n\n", (float) best_tt/FACTOR);

                  }
          else
            fprintf(ofp,"THE BEST TOUR FOUND IS THE FEASIBLE TOUR ABOVE!\n\n");

fprintf(ofp,"The tabu search routine took %.3f seconds.\n\n", duration);

fprintf(ofp,"The search found a total of %d feasible tours.\n", numfeas);
fprintf(ofp,"The search found a total of %d different tours.\n", numdiff);

printf(": The search found %d feasible tours and %d different tours.\n", numfeas, numdiff);

#         ifdef PRINT_HTBL

          fprintf(ofp,"\nHashing Table:\n");
          fprintf(ofp," VAL \t COST  TOURHV  PENALTY  LAST FOUND\n");

          for (i=0; i<HTSIZE; ++i) {
                  if (hashtbl[i] != NULL)  fprintf(ofp,"\n%5d\t", i);

            for (ptr = hashtbl[i]; ptr != NULL; ptr = ptr->next)
                  fprintf(ofp,"%7d%7d%7d%5d %u\n\t", ptr->cost, ptr->tvltime, ptr->pen, ptr->lastfound,
ptr->thval);
          }/*end for i*/
#         endif

/*Free all the memory structures after every problem to start over every time*/
          free(m);
          free(z);
          free(tour);
          free(best_tour);
          free(best_ftour);
          free(oldtour);

          for (i=0; i < nnodes; ++i)             /*Free the time matrix*/
                  free(time[i]);
            free(time);

          for (i=0; i < nnodes; ++i)             /*Free the tabu list*/
                  free(tabu_list[i]);
            free(tabu_list);

          return &soln[0];

}/***********************end of the main program*****************************/


/**This function computes the incremental change in the value of the incumbent
 tour to the proposed neighbor tour.*/

move_delta(i, d, n, t, tt, ti)
          int i;                     /*The starting point for computing the value.*/
          int d;                     /*The depth of the insertion.*/
```

```
        int n;                      /*The number of nodes in the tour.*/
        struct node *t;             /*The incumbent tour structure.*/
        struct node *tt;    /*The neighbor (temporary) tour structure.*/
        int  **ti;          /*The time/distance matrix.*/
{
        int is;                     /*The starting point for the nbr schedule.*/
        int j;                      /*The index of the "target" node of insertion.*/
        int delin;                  /*The incremental tour travel time.*/
        int delout;                 /*The incremental tour travel time.*/

        struct node *npi, *npi1;/*Node pointer indexes used to iterate.*/

        /*These additions make version 4:  Stops computing when a vehicle
        node is encountered after the "within" area of change.*/

        int iend;           /*Index to end of "within" area of insertion.*/
        struct node *npe; /*Pointer to the end of within insertion.*/

        delin = delout = 0;

        if (d>0) {
                j = i+d;
                is = i+d-1;
                iend = i+d+1;
            }/*end if*/
        else {
                j = i+d-1;
                is = i+d;
                iend = i+d+3;
            }/*end else.*/

        npi = &tt[is-1];

        npe = &tt[iend];

/* This is a procedure for updating the schedule from istart to the terminal depot.*/

        while (npi < npe || npi->type != 2) {

          npi1 = npi+1;

          npi1->arr = npi->dep + ti[npi->id][npi1->id];

            if (npi1->type == 2) {
                npi1->dep = npi1->e;
                npi1->wait = 0;   }/*end if vehicle node.*/
            else {
                npi1->dep = max(npi1->e, npi1->arr);
                npi1->wait = npi1->dep - npi1->arr; }/*end if customer node.*/

          ++npi;
        }/*end while*/

        delout = ti[t[i-1].id][t[i].id]+ti[t[i].id][t[i+1].id]
```

```
                        +ti[t[j].id][t[j+1].id];

        delin = ti[t[i-1].id][t[i+1].id]+ti[t[j].id][t[i].id]
                    +ti[t[i].id][t[j+1].id];

        return (delin - delout);

}  /* This ends the computation of the move value.*/


/**This function computes the tour schedule for a neighbor (temporary) tour.**/
 /*** It returns the total value of the tour tour length.****/

tour_sched(istart, n, t, ti)
        int istart;                     /*The starting point for computing the sched.*/
        int n;                          /*The number of nodes in the tour.*/
        struct node *t;                 /*The tour structure.*/
        int  **ti;              /*The time/distance matrix.*/
{

        struct node *h;                 /*Index for the pointer to istart-1.*/
        struct node *lastnp;            /*Index for the pointer to the last node.*/
        struct node *np, *np1;          /*Node pointer indexes used to iterate.*/
        int tour_length;   /*The total tour length.*/

        tour_length = 0;
        h = &t[istart-1];
        lastnp = &t[n-1];

/*This computes the tour length from the origin depot to istart.*/

        for (np = &t[0]; np < h; ++np)
          tour_length +=  ti[np->id][(np+1)->id] + (np+1)->wait;

/* This is a procedure for updating the schedule from istart to the terminal depot.*/

        for (np = h; np < lastnp; ++np) {

          np1 = np+1;

          np1->arr = np->dep +ti[np->id][np1->id];

          if (np1->type == 2) {
                np1->dep = np1->e;
                np1->wait = 0;     }/*end if vehicle node.*/
          else {
                np1->dep = max(np1->e, np1->arr);
                np1->wait = np1->dep - np1->arr; }/*end if customer node.*/

          tour_length +=  ti[np->id][np1->id] + np1->wait;

         }/*end for*/

         return (tour_length);
```

```
} /* This ends the computation of the tour schedule.*/


/*******This function computes the EXACT infeasibility penalty.***************/

comp_timepen(n, t)
        int n;                    /* The number of nodes.*/
        struct node *t;           /* The tour pointer.*/
{
        struct node *np;  /*The index for the node pointer.*/
        struct node *lastnp;      /*The index for pointer to the last node.*/
        int infeas;               /*The total infeasibility of the tour.*/
        int num;          /* The number of infeasible nodes.*/
        int y;                    /* Dummy.*/

        infeas = num = 0;

        lastnp = &t[n-1];

        for (np = &t[0]; np <= lastnp ; ++np)  {
          y = np->dep - np->l;
          if (y > 0) {
            ++num;
            infeas += y;
          } /* end if*/
        } /*end for*/
        return infeas;
}                         /*This ends the comp_penalty function.*/
/**************Computes route penalty**********/
comp_routepen(n,rpen,t,k,ofp)
        int n;                    /* The number of nodes in tour.*/
        float *rpen;              /* The tour penalty*/
        struct node *t; /*The current tour*/
        int k;

{

        float routepen;           /*The total penalty of the tour.*/
        float y;
        float z;
        float q;
        float bb;
        float zz;/*dummy variable*/
   int i;
        int j;/*counter*/
   int x;
        int aa;
        float *rp;
   rp = (float *)calloc(n, sizeof(float));
        routepen = 0.0;
        z=0.0;
        y=0.0;
```

```
    q=1.0;
    bb=0.0;
    zz=0;
    x=0;
    aa=1;

    /*for (i = 1; i <= n-1; ++aa)  {
            rp[i]=0;
    }*/



    for (i = 1; i <= n-1; ++i)  {
            if(t[i].type == 1) {
            x=t[i].id;

            rp[aa]=rpen[x];
            aa+=1;


                    /*printf("\npenalty for node is %f\n",rpen[x]);*/

            }
            else {

                    /*q=min(q,y);

                    routepen=(int)(1000.0*(1-q));*/

            if (k>=6000)
            routepen=0;
    else{

    z=z+(rp[1]+rp[1]*rp[2]+rp[1]*rp[2]*rp[3]+rp[1]*rp[2]*rp[3]*rp[4]+rp[1]*rp[2]*rp[3]*rp[4]*rp[
5]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]+rp[1]*rp[2]*rp[3]*rp
[4]*rp[5]*rp[6]*rp[7]*rp[8]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]+rp[1]*rp[2]*rp[3]*r
p[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10
]*rp[11]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]+rp[1]*rp[2]*rp[3]
*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]
*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8
]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp
[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]
*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*r
p[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]+rp[1]*rp[2]*rp[3]*r
p[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp
[19]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]
*rp[16]*rp[17]*rp[18]*rp[19]*rp[20]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[
11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*rp[20]*rp[21]+rp[1]*rp[2]*rp[3]*rp[4]*r
p[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*r
p[20]*rp[21]*rp[22]+rp[1]*rp[2]*rp[3]*rp[4]*rp[5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13
]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*rp[20]*rp[21]*rp[22]*rp[23]+rp[1]*rp[2]*rp[3]*rp[4]*rp[
5]*rp[6]*rp[7]*rp[8]*rp[9]*rp[10]*rp[11]*rp[12]*rp[13]*rp[14]*rp[15]*rp[16]*rp[17]*rp[18]*rp[19]*rp[
20]*rp[21]*rp[22]*rp[23]*rp[24]);
```

```
         routepen=1000*(1/z);

        aa=1;
        for (j = 1; j <= n-1; ++j)  {
                rp[j]=0;
        }

        /*printf("The iteration is %d",k);*/

        }


                }
        } /*end for*/
        /*fprintf(ofp,"\nroutepenin is %f\n",routepen);*/

        free(rp);
        return routepen;


}                       /*This ends the comp_routepen function.*/

/**************A function to perform a swap move.*****************************/

/* Swaps two node structures in the specified tour.*/

swap_node(i, j, t)
        int i, j;               /*Indices of the nodes to be swapped.*/
        struct node *t;                 /*Structure for the tour to be swapped.*/
{
        struct node x;                  /*Temporary variable for the SWAP macro.*/
        struct node *temp;              /*Temporary tour pointer.*/

        SWAP(t[i], t[j], x);

}/*end of swap node function*/

/**************A function to perform an insertion move.**********************/

/* Performs "depth" number sequence of swaps and returns a pointer to the resulting tour structure
vector.*/

struct node *insert (is, t, depth, n)
        int is;                         /*The node to be inserted.*/
        int depth;                      /*The depth of the insertion (depth> 0) =>
                                         later; (depth<0) => earlier in the tour.*/
        int n;                          /*The number of nodes in the tour.*/
        struct node *t;                 /*The current tour pointer.*/
{
        int j, i;               /*Indices for counting.*/
        struct node x;                  /*Temporary variable for the SWAP macro.*/
        struct node *t_t,*pt;           /*The structure for the new "inserted" tour.*/
        t_t = (struct node *)calloc(n, sizeof(NODE));
        pt = t_t;

        for (i=0; i< n; ++i)            /*Make the temp tour = incumbent tour.*/
          t_t[i] = t[i];

        if (depth > 0)  {
          for (j=0; j < depth; ++j)
                SWAP(t_t[is+j], t_t[is+j+1], x);
        } /*endif*/

        else {
          for (j= 0; j > depth; --j)
                SWAP(t_t[is+j], t_t[is+j-1], x);
        } /*end else*/

        free(t);

        return pt;

} /* end of the Insertion function. */

/***********A Function to compute the sum of the waiting time.****************/

sum_wait(n, t)
        struct node *t;                 /*The tour to be printed.*/
        int n;                          /*The number of nodes in the tour.*/
{
        struct node *lastnp;            /*Index for the pointer to the last node.*/
        struct node *np;   /*Node pointer indexes used to iterate.*/
        int sum;           /*The waiting time sum.*/

        sum = 0;
        for (np = &t[0]; np <= &t[n-1]; ++np)
                sum += np->wait;

        return sum;

}/*The end of the sum_wait function.*/
```

VI. BIBLIOGRAPHY

Bodin, L., Golden B., Assad A, and Ball M. "Routing and Scheduling of Vehicles and Crews: The State of the Art", *Computers and Operations Research*, Vol 10 No: 2, 1983

Carlton, William, "A TABU Search to the General VRP", Dissertation, University of Texas, Dec. 1995

Department of Reconnaissance Operations (DARO), MAE UAV Program, www.acq.osd.mil/daro/uav/mae.html, July 4 1996

Desrochers, M., J. Desrosiers, and M. Solomon, "A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows", *Operations Research*, Vol. 40, No. 2, pp. 342-353, 1992

Dumas, Y., J. Desrosiers, E. Gelinas, and M. Solomon, "An Optimal Algorithm for the Traveling Salesman Problem with Time Windows", *Operations Research*, Vol. 43, No. 2, pp.367-371, 1995

Eilon, S., C. D. T. Watson-Tgandy, and N. Christofides, "Distribution Management: Mathematical Modeling and Practical Analysis", New York, Hafner Publishing Company, 1971

Glover F. "TABU Search--Part I", *ORSA Journal on Computing*, Vol 1. No.3, Summer 1989

Glover, F., "TABU Search—Part 2", *ORSA Journal on Computing*, Vol1, No. 3, pp. 190-206, 1990a

Glover, F., "TABU Search: A Tutorial", *Interfaces*, Vol 20, No. 4. pp. 266-273, 1990

Gonsalvez, David, Nicholas Hall, and others, "Heuristic Solutions and Confidence Intervals for the Multicovering Problem". *European Journal of Operational Research* No.31, pp. 94-101, 1987

Grier, Peter. "Darkstar and Its Friends", *AIR FORCE Magazine*, Jul 1996

Hewish, Mark. "Sensor Payloads for Unmanned Aerial Vehicles", *International Defense Review*: pp. 53 Vol. 28 No.12 December 1,1995

Janes, "Self-Propelled SAMS", *Ground Based Weapon Systems*, Jane's Information Group Inc. Alexandria, Virginia, USA, 1994

Lenstra J. K. Rinnooy Kan A. H. G., "Complexity of Vehicle Routing and Scheduling Problems", *Networks*, Vol. 11, pp. 221-227, 1981

Laporte, G. , "The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms", *European Journal of Operational Research*, Vol. 59, pp.231-247, 1992

Mass, Robert C., GOR/SM/75-10, "A Computer Simulation of an RPV (remotely piloted vehicle) Strike Mission". *AFIT Thesis*, Feb 1976

Pearson, Major, USAF, Project Officer, "Air Combat Command Concept of Operations for Endurance Unmanned Aerial Vehicles". OPR ACC/XPJC, Langley AFB VA 23665-2778, Draft, Aug 95

Secretary of the Air Force Policy Letter, Aug 12, 1996

Savelsbergh, M.W.P., "The Vehicle Routing Problem with Time Windows: Minimizing Route Duration", *ORSA Journal on Computing*, Vol 44, No 2, pp. 146-154, 1992

Taillard E D., Laporte, Gendreau., "Vehicle Routing with Multiple Use of Vehicles", *Journal of the Operational Research Society*, Vol 47, No. 8, 1996

VII. Vita

Mark R. Sisson [redacted], the son of Tommy Wendell Sisson, and Karna [redacted] Sisson. In December 1983 he graduated from Auburn University of Montgomery, in Alabama, earning a Bachelor of Science in Mathematics. After graduation, he accepted a commission in the United States Air Force. While serving on active duty, he enrolled in Texas Christian University and earned a Masters of Liberal Arts in July of 1988. He went on to become qualified in the B-52G/H, instructor and evaluator qualified on the RC-135V/W, and Operations Officer of EW/ETSS for the KE-3B. He entered the Graduate School of Engineering, Air Force Institute of Technology August of 1995.

[redacted]

[redacted]

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE February 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE APPLYING TABU HEURISTIC TO WIND INFLUENCED, MINIMUM RISK, AND MAXIMUM EXPECTED COVERAGE ROUTES	5. FUNDING NUMBERS
6. AUTHOR(S) Mark R. Sisson, Major, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology/ENS 2750 P Street Wright-Patterson AFB, Ohio 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GOR/ENS/97-06
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ACC/XP-SAS 204 Dodd Blvd, Ste 202 Langley AFB VA 23665-2778	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution is Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The purpose of this thesis is to provide Air Combat Command a method for determining the number of predator unmanned aerial vehicles (UAVs) required to cover a pre-selected target.

Extending previous research that employs reactive TABU search methods for deterministic vehicle routing problems, this thesis incorporates wind effects that can significantly alter the travel times for any given scenario. Additionally, it accounts for possible attrition by introducing minimum risk route and expected number of target covered to the objective function. The results of the TABU search and subsequent monte-carlo simulation: gives the number of predator's required to cover a target set, identifies "robust" routes, and suggests routes that increase expected number of targets covered while reducing losses.

14. SUBJECT TERMS TABU Heuristic, Monte-Carlo Simulation, Expected nodes covered, mTSPTW, Traveling Salesman Problem, Predator UAV	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL